**Дипломная работа: Анализ и сравнение написания web-приложений с использованием разных фреймворков:**разработать простые веб-приложения с использованием Django, Flask и FastAPI, провести их сравнение.

Автор: Франтенков Александр Валерьевич

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 2](#_Toc181181670)

[ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 4](#_Toc181181671)

[1.1 Основные понятия и определения 4](#_Toc181181672)

[1.2. Основные аспекты синтетической речи, задачи генерации и классификации 6](#_Toc181181673)

[1.3. Обзор популярных фреймворков для разработки web-приложений на Python 9](#_Toc181181674)

[1.4. Выбор инструментов и моделей для классификации и генерации речи 14](#_Toc181181675)

[1.5. Описание требований к системе и архитектуре решения 17](#_Toc181181676)

[ГЛАВА 2. ПРОЕКТИРОВАНИЕ ВЕБ-ПРИЛОЖЕНИЙ 21](#_Toc181181677)

[2.1. Планирование и анализ требований к веб-приложениям 21](#_Toc181181678)

[2.2. Обучение модели классификации речи 28](#_Toc181181679)

[2.2. Проектирование компонентов приложения 39](#_Toc181181680)

[ГЛАВА 3. РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ 43](#_Toc181181681)

[3.1. Реализация с использованием Django 43](#_Toc181181682)

[3.2. Разработка на Flask 49](#_Toc181181683)

[3.3. Разработка с использованием FastAPI 55](#_Toc181181684)

[3.4. Сравнительный анализ реализации приложения 61](#_Toc181181685)

[ЗАКЛЮЧЕНИЕ 65](#_Toc181181686)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 67](#_Toc181181687)

[ПРИЛОЖЕНИЯ 69](#_Toc181181688)

[Приложение А 69](#_Toc181181689)

[Приложение Б 74](#_Toc181181690)

[Приложение В 77](#_Toc181181691)

# ВВЕДЕНИЕ

Тема разработки веб-приложений остается одной из наиболее востребованных в условиях растущего числа цифровых сервисов и приложений, ориентированных на взаимодействие с пользователем. Для создания высокопроизводительных, масштабируемых и надежных веб-приложений разработчикам необходимо тщательно подходить к выбору технологий, в частности, фреймворков, которые определяют структуру и функциональность системы. Среди множества решений на языке программирования Python значительное место занимают такие фреймворки, как Django, Flask и FastAPI. Эти инструменты обладают различными архитектурными подходами, которые влияют на удобство разработки, производительность, возможности для масштабирования и уровень интеграции с внешними модулями, что особенно актуально при создании специализированных веб-сервисов.

Актуальность исследования определяется необходимостью оптимального выбора фреймворка в зависимости от задач конкретного веб-приложения. Несмотря на широкое распространение каждого из рассматриваемых фреймворков, их возможности, подходы к реализации, а также производительность могут существенно различаться. Задача выбора подходящего фреймворка остается актуальной для разработчиков, поскольку определяет скорость разработки, удобство сопровождения и возможности дальнейшего развития системы. Проведенный анализ и сравнение помогут выявить сильные и слабые стороны Django, Flask и FastAPI, что позволит более обоснованно подходить к выбору технологий для новых проектов.

Объект исследования в работе – это процесс создания веб-приложений с применением различных фреймворков, обеспечивающих различный подход к построению архитектуры, интеграции компонентов и управлению ресурсами.

Предмет исследования – сравнительный анализ фреймворков Django, Flask и FastAPI на предмет их производительности, архитектурных решений, гибкости и возможностей для расширения, а также удобства работы для разработчиков. В работе также рассматривается практическое применение каждого из фреймворков для задачи классификации и генерации синтетической речи, что позволяет продемонстрировать их особенности на реальной задаче.

Целью исследования является сравнительный анализ Django, Flask и FastAPI с целью выявления оптимального фреймворка для разработки веб-приложений, решающих конкретные задачи по классификации и генерации синтетической речи. Исследование направлено на определение ключевых параметров производительности, гибкости и масштабируемости, которые играют важную роль при выборе технологии для веб-разработки.

Для достижения этой цели были поставлены следующие задачи:

1. Исследовать предметную область и выявить ключевые особенности каждого фреймворка;
2. Сформировать и обосновать критерии для анализа и сравнения фреймворков;
3. Реализовать веб-приложение с применением каждого фреймворка для задачи классификации и генерации синтетической речи;
4. Проанализировать полученные результаты и определить сильные и слабые стороны каждого из подходов;
5. Сформулировать рекомендации по выбору фреймворка в зависимости от особенностей проекта.

Практическая ценность работы заключается в предоставлении сравнительного анализа, который поможет разработчикам более эффективно выбирать инструменты для создания веб-приложений. Использование примера задачи классификации и генерации синтетической речи позволяет оценить, насколько эффективно каждый фреймворк справляется с реальными нагрузками и обработкой данных.

# ГЛАВА 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

## 1.1 Основные понятия и определения

1. Фреймворк (Framework): программная платформа, которая предоставляет структуру, инструменты и библиотеку готовых компонентов для разработки приложений. В контексте данной работы рассматриваются три фреймворка для веб-разработки: Django, Flask и FastAPI. Каждый из них обладает уникальными свойствами, которые влияют на структуру приложения, производительность и удобство разработки.

2. Веб-приложение (Web Application): программное приложение, выполняющееся на сервере и доступное пользователям через веб-браузер. В данной работе веб-приложение используется для реализации задач классификации и генерации синтетической речи с целью сравнения эффективности разных фреймворков.

3. Производительность (Performance): показатель эффективности работы веб-приложения, включающий скорость обработки запросов, время отклика и стабильность работы под нагрузкой. Производительность является одним из ключевых критериев для оценки фреймворков в данной работе.

4. Масштабируемость (Scalability): способность веб-приложения адаптироваться к изменению объема данных и увеличению нагрузки, сохраняя стабильную производительность. Масштабируемость зависит от возможностей фреймворка и его архитектуры.

5. Асинхронность (Asynchronous Processing): подход к обработке запросов, позволяющий приложению обрабатывать несколько задач одновременно, не дожидаясь завершения каждой. Асинхронность помогает ускорить работу приложений и является отличительной чертой FastAPI.

6. API (Application Programming Interface): интерфейс программирования приложений, предоставляющий доступ к функциям приложения и взаимодействие между сервисами. В данной работе API используется для создания маршрутов обработки аудиофайлов и текстов, что позволяет пользователю взаимодействовать с веб-приложением.

7. Классификация речи (Speech Classification): задача машинного обучения, направленная на определение типа речи, например, синтетической или реальной. В данной работе классификация речи используется в качестве примера задачи, решаемой веб-приложением для демонстрации возможностей различных фреймворков.

8. Генерация речи (Speech Generation): процесс синтеза речи на основе текстового ввода. Генерация синтетической речи помогает иллюстрировать функциональные возможности каждого из фреймворков для реализации сложных вычислительных задач в реальном времени.

9. Предобученная модель (Pre-trained Model): модель машинного обучения, обученная на большом наборе данных и используемая для решения задач без необходимости дополнительного обучения. В данной работе используются предобученные модели для классификации и генерации речи, что упрощает интеграцию в веб-приложение и позволяет сосредоточиться на сравнении фреймворков.

10. JWT (JSON Web Token): стандарт передачи данных в формате JSON для аутентификации пользователей. JWT позволяет безопасно передавать токены доступа между клиентом и сервером, обеспечивая защиту данных пользователей. В работе JWT используется для реализации механизма аутентификации в каждом из фреймворков.

11. ORM (Object-Relational Mapping): методика, позволяющая работать с базой данных через объектно-ориентированные конструкции, представляя данные в виде объектов. Django и FastAPI активно используют ORM для работы с базой данных, упрощая взаимодействие с данными для разработчика.

Эти термины помогают понять ключевые аспекты разработки веб-приложений на разных фреймворках, ориентированных на задачи обработки данных и взаимодействия с пользователем.

## 1.2. Основные аспекты синтетической речи, задачи генерации и классификации

Синтетическая речь — это звуковой сигнал, сгенерированный алгоритмами, воспроизводящими особенности человеческой речи. В последние десятилетия синтез речи нашел широкое применение в самых разных областях: от персональных голосовых ассистентов до автоматизированных телефонных систем. Подобные приложения стали возможны благодаря развитию машинного обучения, технологий обработки естественного языка и анализу аудио. Современные модели синтеза речи позволяют добиться настолько высокого уровня реализма, что синтетические аудиофайлы практически невозможно отличить от настоящих. Это создает и ряд положительных возможностей, и новые вызовы. С одной стороны, синтетическая речь активно используется в коммерческих продуктах и обеспечивает доступ к информации людям с ограничениями по слуху или зрению. С другой стороны, она может стать инструментом для создания дезинформации, если не будет разработано адекватных мер контроля, включая алгоритмы различения настоящей и синтетической речи.

В данной работе синтетическая речь используется в качестве примера задачи, решаемой с применением различных фреймворков для создания веб-приложений. Основная цель приложения — это не только демонстрация возможностей каждого фреймворка, но и сравнение их производительности, гибкости и удобства в процессе интеграции алгоритмов для работы с речью. В этом контексте важно выделить две ключевые задачи: генерацию синтетической речи и её классификацию.

*Генерация синтетической речи*

Генерация синтетической речи представляет собой сложный процесс, в ходе которого текстовая информация преобразуется в звуковой сигнал, максимально приближенный к человеческой речи. На сегодняшний день синтез речи активно развивается, и появление новых моделей, таких как нейронные сети, сделало его более доступным и реалистичным. Классические подходы синтеза, такие как формантный синтез или синтез с использованием вокодеров, во многом были вытеснены методами глубокого обучения, например, архитектурами, построенными на основе рекуррентных и сверточных нейронных сетей (например, WaveNet, Tacotron и другие). Эти архитектуры обеспечивают высокую точность передачи интонации, акцентирования и других особенностей речи, которые делают синтетический голос естественным и реалистичным.

Для синтеза речи, используемого в данной работе, применялась модель Silero TTS, работающая с аудиофайлами на русском языке. Silero TTS — это модель, обученная на данных из открытых источников, которая позволяет сгенерировать речь высокого качества, передавая все интонационные и фонетические особенности. Применение предобученной модели позволяет избежать затратного процесса обучения и в значительной мере ускоряет процесс разработки приложения [4].

Несмотря на то, что сгенерированная речь может казаться совершенно естественной, существуют некоторые особенности синтетических аудиофайлов, которые можно выявить с помощью анализа звуковых характеристик. Эта возможность раскрывается при переходе ко второй задаче — классификации синтетической речи.

*Классификация речи: задача распознавания синтетических сигналов*

Классификация речи заключается в том, чтобы определить, является ли данный аудиосигнал результатом естественной речи или синтезированным образцом. Данный процесс может оказаться полезным в различных сценариях: от повышения уровня безопасности (например, в банкинге или безопасности данных) до выявления возможных случаев дезинформации. Классификация речи основана на анализе звуковых признаков, которые могут отличать синтетические и естественные сигналы. Для выполнения классификации применяются модели машинного обучения, использующие заранее извлеченные признаки звуковых сигналов. В качестве таких признаков, например, выступают MFCC-коэффициенты.

*MFCC-признаки и их роль в классификации речи*

Mel-frequency cepstral coefficients (MFCC) — это числовые характеристики звука, которые позволяют анализировать звуковые сигналы с точки зрения восприятия частот человеческим слухом. MFCC-признаки используются во многих задачах обработки речи, включая распознавание речи, идентификацию говорящих и классификацию аудиосигналов.

MFCC-коэффициенты являются результатом преобразования аудиосигнала с использованием мел-шкалы частот. Мел-шкала была разработана специально для моделирования восприятия звука человеческим ухом, что делает MFCC-признаки особенно подходящими для задач анализа речи. Извлечение MFCC-признаков включает несколько этапов:

* Преобразование аудиосигнала в короткие временные сегменты (или окна) для анализа временных характеристик звука.
* Преобразование Фурье для каждого окна, что позволяет перейти от временного представления сигнала к частотному.
* Применение мел-фильтров, представляющих собой набор полос частот, соответствующих восприятию звука человеком.
* Преобразование логарифмических амплитуд с помощью косинусного преобразования, результатом которого становятся MFCC-коэффициенты.

Эти признаки представляют собой числовое описание частотной структуры аудиосигнала, учитывая особенности человеческого восприятия звука. Для задач классификации MFCC-признаки обеспечивают удобный и информативный формат данных, который позволяет алгоритмам машинного обучения отличать синтетические образцы от реальных. Например, сгенерированная искусственная речь может отличаться по уровню плавности переходов, диапазону частот и другим параметрам, что будет учтено в значениях MFCC-коэффициентов.

*Применение MFCC-признаков в работе*

В контексте данной работы MFCC-признаки извлекаются из аудиофайлов, загруженных пользователем в веб-приложение, реализованное на фреймворках Django, Flask и FastAPI. Эти признаки используются в качестве входных данных для модели машинного обучения, которая решает задачу классификации речи. Сначала аудиофайл обрабатывается с целью извлечения MFCC-коэффициентов, после чего полученные данные передаются в классификационную модель, которая определяет тип речи. Результат классификации выводится в интерфейсе приложения, что позволяет пользователю получить мгновенную обратную связь о том, является ли речь реальной или синтетической.

MFCC-признаки играют важную роль в процессе классификации, обеспечивая приложение данными, которые наиболее полно отражают звуковые характеристики аудиосигнала. Их использование позволяет с высокой точностью идентифицировать синтетическую речь, что важно для задач безопасности и контроля качества данных. В ходе работы MFCC-признаки также демонстрируют важность выбора подходящих инструментов и архитектуры, так как каждая часть процесса — от генерации синтетической речи до её классификации — требует значительных вычислительных ресурсов. Это, в свою очередь, позволяет объективно оценить производительность каждого фреймворка в условиях реальной задачи, что составляет основу для дальнейшего сравнительного анализа Django, Flask и FastAPI.

## 1.3. Обзор популярных фреймворков для разработки web-приложений на Python

**Django: особенности и преимущества для веб-разработки**

Django — высокоуровневый фреймворк для веб-разработки на Python, который предоставляет обширный набор инструментов для создания полноценных приложений. Одной из ключевых особенностей Django является принцип «все включено», что означает, что фреймворк содержит встроенные решения для работы с базами данных, управления аутентификацией, обработки форм и других важных элементов веб-приложения. Основу Django составляет архитектура MVT (Model-View-Template), аналогичная модели MVC, которая четко разделяет логику данных, контроллеры и отображение. Модели (Models) отвечают за работу с базой данных, представления (Views) — за обработку запросов, а шаблоны (Templates) представляют собой HTML-код, определяющий вид страницы для пользователя [1].

Одним из главных преимуществ Django является встроенная ORM (Object-Relational Mapping), позволяющая работать с базой данных с помощью Python-классов, что освобождает разработчиков от написания SQL-запросов и делает код приложения более чистым и удобным для сопровождения. Важной частью Django является встроенная система администрирования, которая автоматически создает административную панель для управления данными и позволяет добавлять, редактировать или удалять записи в базе данных без дополнительной разработки интерфейсов. Это особенно полезно для бизнес-приложений, где требуется оперативное управление контентом.

Django также включает мощные встроенные механизмы для обеспечения безопасности, такие как защита от XSS-атак, CSRF, SQL-инъекций и других распространенных угроз, что делает его надежным выбором для приложений, где важна безопасность пользовательских данных. Кроме того, фреймворк поддерживает масштабируемость и подходит для создания крупных приложений, таких как социальные сети и информационные порталы. Преимуществом является и обширное сообщество разработчиков, которое поддерживает множество обучающих ресурсов и позволяет найти ответ на практически любой вопрос. В рамках данной работы Django будет использоваться для создания веб-приложения с задачей классификации и генерации синтетической речи, что позволит оценить его производительность и удобство при реализации подобных задач. Архитектура Django представлена на рисунке 1.

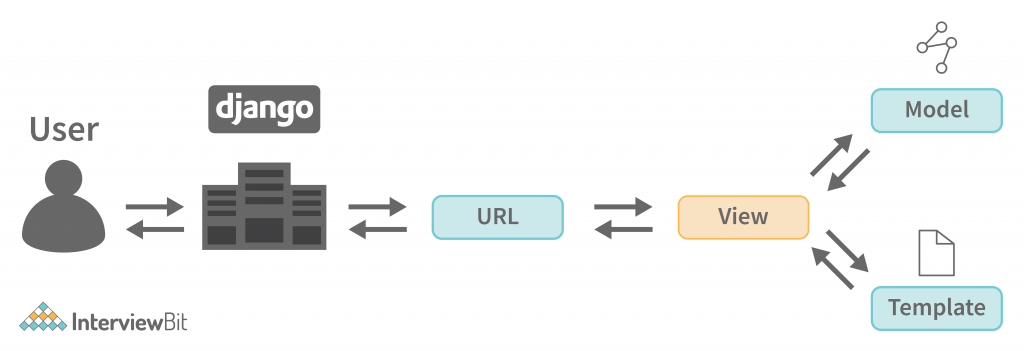


Рисунок 1 – Архитектура Django

**Flask: легковесность и гибкость для REST API**

Flask — это минималистичный и легковесный фреймворк для веб-разработки на Python, ориентированный на создание простых, гибких и масштабируемых приложений. В отличие от Django, который предоставляет множество встроенных инструментов, Flask придерживается философии «минимальной структуры», предлагая разработчикам только основные компоненты для создания веб-приложений, а все остальное — опционально и настраивается по мере необходимости. Такая легковесность делает Flask идеальным для создания REST API и небольших веб-приложений, где требуется высокая гибкость в выборе компонентов и модулей [2].

Flask построен на основе WSGI (Web Server Gateway Interface) и использует встроенный сервер разработки, что упрощает локальное тестирование и отладку. Благодаря простоте своей структуры, Flask позволяет разработчикам добавлять только те библиотеки и компоненты, которые требуются для конкретного проекта, делая приложение менее ресурсоемким и более управляемым. В Flask отсутствует встроенная ORM или система аутентификации, что делает его полностью гибким для интеграции сторонних решений — таких как SQLAlchemy для работы с базой данных или Flask-Login для аутентификации.

Для создания REST API Flask является оптимальным выбором, поскольку позволяет легко настроить маршруты, обрабатывать запросы и управлять JSON-ответами. Его простой, интуитивно понятный синтаксис и отсутствие жестких ограничений делают Flask удобным для прототипирования и быстрого развертывания API, обеспечивая при этом высокую степень контроля над каждым компонентом.

В рамках данной работы Flask будет использоваться для создания API, ориентированного на задачи классификации и генерации синтетической речи. Это позволит оценить его производительность, гибкость в интеграции моделей машинного обучения и удобство работы с REST-запросами, что особенно важно для приложений, где требуется модульная структура и минимальные накладные расходы на дополнительные функции. Пример возможной архитектуры web-приложения приведен на рисунке 2.

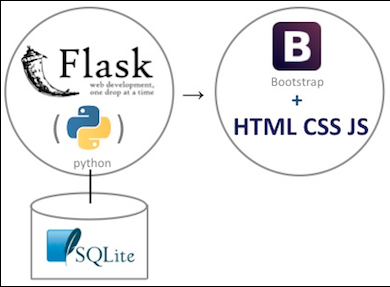


Рисунок 2 – Архитектура Flask

**FastAPI: асинхронное программирование и высокопроизводительные API**

FastAPI — это современный фреймворк для веб-разработки на Python, разработанный специально для создания высокопроизводительных REST API с поддержкой асинхронного программирования. FastAPI использует асинхронные вызовы и поддержку библиотеки ASGI (Asynchronous Server Gateway Interface), что позволяет обрабатывать множество запросов одновременно, минимизируя задержки и делая его особенно подходящим для приложений с высокой нагрузкой. Благодаря этой асинхронности, FastAPI обеспечивает значительно более высокую производительность по сравнению с традиционными фреймворками, такими как Django и Flask [3].

Одной из ключевых особенностей FastAPI является автоматическая генерация документации API. Используя возможности Python type hints, FastAPI создает интуитивно понятный интерфейс для разработчиков и автоматически генерирует документацию в формате OpenAPI и Swagger. Это упрощает тестирование и интеграцию, обеспечивая наглядное представление всех доступных маршрутов и параметров, что удобно при создании сложных API.

Еще одно преимущество FastAPI — удобство работы с JSON-данными, а также валидация и сериализация данных. Благодаря встроенной поддержке Pydantic, FastAPI позволяет автоматически проверять типы данных и обрабатывать ошибки, что упрощает создание надежных API и снижает вероятность ошибок. Как и Flask, FastAPI предоставляет гибкость выбора компонентов, таких как базы данных или аутентификация, что позволяет разработчикам адаптировать его под нужды конкретного проекта.

FastAPI идеально подходит для задач, требующих обработки больших объемов данных или взаимодействия с моделями машинного обучения в реальном времени. В данной работе FastAPI используется для создания высокопроизводительного API, ориентированного на задачи классификации и генерации синтетической речи. Такой подход позволяет проверить, насколько эффективно фреймворк справляется с обработкой асинхронных запросов и обеспечивает ли он нужный уровень производительности при взаимодействии с предобученными моделями и сложными вычислительными процессами. Компоненты приложения на FastApi приведены на рисунке 3.

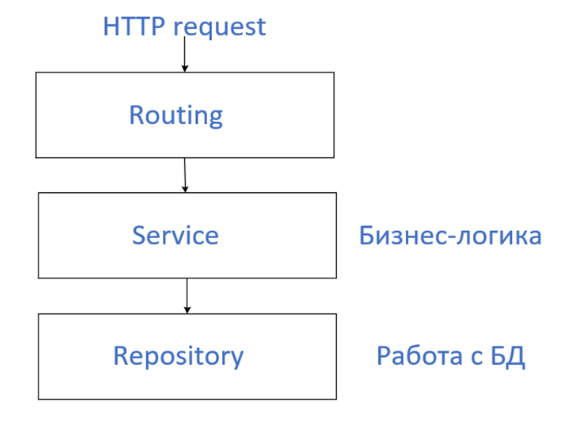


Рисунок 3 – Компоненты приложения FastApi

## 1.4. Выбор инструментов и моделей для классификации и генерации речи

В рамках задачи классификации синтетической речи важным этапом является выбор подходящей предобученной модели, которая способна эффективно различать синтетические и реальные аудиозаписи. При создании веб-приложения с такой функциональностью выбор модели оказывает значительное влияние на качество классификации, производительность приложения и удобство интеграции в архитектуру различных фреймворков, таких как Django, Flask и FastAPI [6].

Для классификации речи был выбран подход на основе нейронной сети, поскольку он демонстрирует высокую точность в задачах аудиоанализа. Одним из ключевых шагов для классификации аудиоданных является предварительное извлечение характеристик звука. В данной работе используется извлечение MFCC-признаков (Mel-Frequency Cepstral Coefficients), которые позволяют преобразовать звуковой сигнал в набор числовых данных, наиболее информативных для классификации. Эти признаки передаются на вход модели и позволяют нейронной сети определить, является ли данный образец синтетическим или реальным.

В качестве основной модели классификации использована архитектура на основе плотных слоев (dense layers), обученная на аудиозаписях с разметкой для двух классов — реальной и синтетической речи. Данная модель была предварительно обучена на аудиоданных, что позволяет ей сразу демонстрировать высокую точность без необходимости дополнительного обучения в процессе интеграции в веб-приложение. Этот выбор обусловлен тем, что модели глубокого обучения с заранее обученными весами позволяют сократить время на подготовку и настройку, а также обеспечивают высокую производительность в реальных условиях работы приложения.

Среди альтернативных моделей, которые также могли бы быть использованы для решения данной задачи, можно выделить модели на основе сверточных и рекуррентных нейронных сетей. Однако для веб-приложений, ориентированных на обработку запросов в реальном времени, использование моделей с плотными слоями оказалось оптимальным выбором, так как они обладают меньшими вычислительными затратами и хорошо подходят для классификации MFCC-признаков.

Таким образом, для реализации задачи классификации синтетической речи в рамках данной работы была выбрана нейронная сеть с предварительно обученными весами, использующая MFCC-признаки для анализа аудиосигналов. Этот подход обеспечивает точность и скорость классификации, необходимые для эффективной работы приложения в реальных условиях, и позволяет фокусироваться на сравнении производительности фреймворков, интегрирующих модель в различных архитектурах.

Для генерации синтетической речи в данной работе используется модель Silero TTS — современное решение для синтеза речи, поддерживающее генерацию качественного аудиосигнала с использованием текстового ввода. Silero TTS является одной из моделей синтеза речи, основанных на нейронных сетях и способных передавать интонации и естественные особенности голоса, что делает её подходящей для задач, требующих реалистичного звучания [7].

**Особенности и преимущества Silero TTS**. Модель Silero TTS разработана с учетом потребностей разработки приложений, работающих в режиме реального времени, и оптимизирована для работы на устройствах с ограниченными вычислительными ресурсами. Она поддерживает множество языков и акцентов, включая русский, что делает её универсальным инструментом для различных приложений. Silero TTS работает на основе нейронных архитектур, способных обрабатывать текст и преобразовывать его в аудиоформат с высокой точностью передачи интонационных и фонетических нюансов. Модель легко интегрируется в проекты на Python благодаря поддержке библиотеки Torch Hub, что упрощает её использование в веб-приложениях на базе фреймворков Django, Flask и FastAPI.

**Интеграция модели и этапы генерации речи**. Процесс генерации речи с использованием Silero TTS состоит из нескольких этапов. Сначала текстовый ввод от пользователя поступает в веб-приложение, где передается в модель для обработки. Модель генерирует аудиосигнал, учитывая заданные параметры, такие как частота дискретизации, голосовой профиль (например, мужской или женский голос) и прочие настройки, которые придают звучанию нужные акценты и естественные интонации. Полученный аудиофайл сохраняется на сервере и становится доступным для воспроизведения или загрузки пользователем. В данной работе Silero TTS используется для создания аудиофайлов, демонстрирующих возможности каждого из фреймворков по обработке запросов в реальном времени и обработке вычислительно сложных задач.

**Преимущества использования Silero TTS** для задач синтеза речи заключаются в её высокой производительности и простоте интеграции. Синтез речи в реальном времени и возможность гибкой настройки параметров делают её подходящей для приложений, в которых требуется мгновенный отклик и поддержка различных языков и голосов. В рамках работы Silero TTS демонстрирует, насколько эффективно каждый фреймворк (Django, Flask, FastAPI) справляется с задачей генерации речи и обеспечением быстрого доступа к сгенерированному контенту.

## 1.5. Описание требований к системе и архитектуре решения

При разработке веб-приложения для классификации и генерации синтетической речи с использованием фреймворков Django, Flask и FastAPI важно определить основные функциональные и технические требования, чтобы обеспечить стабильную работу и удовлетворить задачи пользователей.

**Функциональные требования**

1. Регистрация и аутентификация пользователей. Приложение должно поддерживать регистрацию новых пользователей и авторизацию существующих, чтобы ограничить доступ к функционалу для незарегистрированных пользователей. Аутентификация должна быть безопасной, включая использование шифрования паролей и токенов для управления сессиями.
2. Загрузка и обработка аудиофайлов для классификации. Пользователь должен иметь возможность загружать аудиофайлы в формате .wav для последующей обработки. Приложение должно принимать загруженные файлы, извлекать из них признаки MFCC и передавать их в модель классификации для определения типа речи (реальная или синтетическая).
3. Классификация аудиофайлов. Приложение должно обеспечивать работу предобученной модели классификации для аудиофайлов. После обработки данных модель определяет, является ли загруженный аудиофайл реальной или синтетической речью, а результаты классификации отображаются пользователю.
4. Генерация синтетической речи. Приложение должно предоставлять пользователю возможность вводить текст для преобразования в аудио с использованием модели Silero TTS. Сгенерированный аудиофайл должен быть доступен для скачивания и воспроизведения в интерфейсе приложения.

**Технические требования**

1. Производительность и масштабируемость. Приложение должно обеспечивать эффективную обработку запросов, включая классификацию и генерацию речи. Для этого важно поддерживать асинхронную обработку задач (особенно в случае FastAPI) и оптимизировать выполнение API-запросов для минимизации задержек.
2. Интеграция предобученных моделей. Приложение должно поддерживать работу с предобученными моделями классификации и генерации речи. Эти модели должны быть интегрированы таким образом, чтобы данные (аудиофайлы и текст) могли быть легко переданы для обработки и возвращены пользователю после выполнения задачи.
3. Безопасность данных. Приложение должно обеспечивать защиту пользовательских данных, включая безопасное хранение учетных данных и использование токенов для управления сессиями. Это важно для защиты данных пользователей и предотвращения несанкционированного доступа к функционалу приложения.
4. Поддержка и управление файлами. Приложение должно поддерживать безопасную загрузку и хранение аудиофайлов, а также обеспечивать доступ к сгенерированным аудиофайлам. Для этого необходимо корректно настраивать директории хранения файлов и их маршрутизацию для скачивания.

Архитектура веб-приложения, предназначенного для задач классификации и генерации синтетической речи, строится вокруг интеграции предобученных моделей и обеспечения эффективной работы API. В данной работе используется гибкая архитектура, которая позволяет реализовать функциональные возможности приложения на разных фреймворках — Django, Flask и FastAPI. Каждый из фреймворков имеет свои особенности в архитектуре, однако все три обеспечивают работу с основными компонентами приложения, такими как API, модели машинного обучения и системы управления файлами.

**Компонент аутентификации и авторизации**. Этот компонент обеспечивает безопасность доступа к функционалу приложения, включая классификацию и генерацию аудиофайлов. Аутентификация пользователей реализуется с использованием токенов (например, JWT), что позволяет ограничить доступ к API и защитить данные пользователей.

**Интеграция предобученной модели классификации речи**. Для задачи классификации используется предобученная модель, которая принимает на вход извлеченные из аудиофайла MFCC-признаки и определяет, является ли речь синтетической или реальной. Модель интегрируется через библиотеку машинного обучения, такую как TensorFlow или PyTorch, и запускается на сервере, где развернуто приложение. Вычисления выполняются на серверной стороне, а результаты классификации передаются пользователю через API.

**Интеграция модели генерации речи (Silero TTS)**. Этот компонент отвечает за генерацию речи на основе текстового ввода от пользователя. Модель Silero TTS подключена через Torch Hub, что позволяет легко загружать её предобученные веса и генерировать аудиофайлы. Введенный пользователем текст преобразуется в аудиоформат, который затем сохраняется на сервере и предоставляется пользователю для скачивания или воспроизведения.

**API для взаимодействия с клиентом**. API представляет собой основной интерфейс, через который происходит обмен данными между клиентом (пользовательским интерфейсом) и сервером. API поддерживает работу с запросами на загрузку аудиофайлов, классификацию, генерацию речи и скачивание сгенерированных файлов. В зависимости от используемого фреймворка, API может быть синхронным (в Django и Flask) или асинхронным (в FastAPI), что позволяет оптимизировать обработку запросов и сократить время отклика.

**Система управления файлами**. Приложение должно обрабатывать загружаемые пользователем аудиофайлы и сгенерированные модели аудиофайлов. Для этого создаются отдельные директории для временного хранения загруженных данных и генерации аудиофайлов. Файлы загружаются на сервер, обрабатываются моделью, после чего пользователь может получить к ним доступ через API.

API приложения поддерживает следующие основные маршруты:

/upload — для загрузки аудиофайлов пользователем. Загруженный файл сохраняется на сервере и передается на классификацию.

/classification — для вызова модели классификации, которая обрабатывает загруженный аудиофайл и возвращает результат классификации (реальная или синтетическая речь).

/generate\_audio — для генерации аудиофайла на основе текстового ввода. Введенный текст передается в модель Silero TTS, которая создает аудиофайл, доступный для скачивания.

/static/uploads и /static/generated\_audio — для предоставления пользователю доступа к сохраненным и сгенерированным файлам.

API каждого фреймворка реализуется с учетом особенностей его структуры. Например, в Django и Flask используются синхронные запросы и традиционная маршрутизация, тогда как FastAPI поддерживает асинхронные запросы, что позволяет обрабатывать несколько запросов параллельно и повышает производительность.

Архитектура приложения обеспечивает гибкость и модульность, позволяя подключать новые компоненты или заменять существующие без кардинальной переработки системы. В то же время использование различных фреймворков позволяет адаптировать API и интеграцию моделей в зависимости от особенностей каждого из них. Django, например, хорошо подходит для управления сессиями и предоставления административного интерфейса, Flask — для гибкой настройки API, а FastAPI — для высокопроизводительных асинхронных запросов.

# ГЛАВА 2. ПРОЕКТИРОВАНИЕ ВЕБ-ПРИЛОЖЕНИЙ

## 2.1. Планирование и анализ требований к веб-приложениям

Этап проектирования веб-приложения начинается с детального анализа функциональных и технических требований, которые определяют, как приложение будет удовлетворять потребности пользователей и обеспечивать устойчивость работы в условиях реальных нагрузок. Для успешной реализации задач классификации и генерации речи, интеграции предобученных моделей и безопасного взаимодействия с пользователем необходимо учесть основные требования к функциональности и безопасности.

**Основные требования к функционалу**

1. **Регистрация и аутентификация пользователей**. Одним из главных функциональных требований является поддержка системы регистрации и аутентификации, которая позволит идентифицировать каждого пользователя и предоставить ему индивидуальный доступ к функционалу приложения. Регистрация новых пользователей должна обеспечивать сохранение основных данных, таких как имя, фамилия, номер телефона и пароль, с обязательной проверкой уникальности телефона для предотвращения дублирования учетных записей. Аутентификация пользователей реализуется с помощью защищенного ввода логина и пароля, а также использования токенов для управления сессиями, что позволяет ограничить доступ к функциям приложения только для зарегистрированных пользователей.
2. **Загрузка и обработка аудиофайлов для классификации**. Важной частью функционала приложения является возможность загрузки аудиофайлов, которые пользователь может предоставить для анализа. Приложение должно принимать файлы, ограниченные по формату (например, .wav), и обрабатывать их для последующей классификации. После загрузки файл сохраняется на сервере, где происходит его предварительная обработка: извлекаются признаки, такие как MFCC-коэффициенты, которые необходимы для работы модели классификации. После выполнения анализа результат классификации отображается пользователю.
3. **Классификация аудиофайлов**. Приложение должно обеспечивать надежную работу модели классификации, которая способна различать синтетическую и реальную речь. Классификация выполняется с использованием предобученной модели, загружаемой из локального хранилища, что позволяет минимизировать задержки и обеспечить быстрый отклик. Для этого приложение должно иметь доступ к вычислительным ресурсам, способным обрабатывать аудиофайлы с минимальной задержкой, и выводить результат, понятный для пользователя — в виде текста или визуального индикатора.
4. **Генерация синтетической речи**. Приложение должно предоставлять возможность генерации синтетической речи на основе текстового ввода пользователя. Пользователь может ввести текст, который будет преобразован в аудиоформат с использованием модели Silero TTS. Сгенерированный файл сохраняется на сервере и становится доступен для скачивания. Этот функционал требует поддержки библиотеки torch для работы модели Silero, а также стабильной производительности сервера для обработки запросов на генерацию в режиме реального времени.
5. **Маршрутизация и интерфейс API**. Для того чтобы функциональные возможности приложения были доступны через пользовательский интерфейс и внешние интеграции, необходимо предоставить доступ к API. API позволяет взаимодействовать с основными функциями приложения, включая маршруты для загрузки файлов, выполнения классификации, генерации речи и скачивания аудиофайлов. Интерфейс API должен быть интуитивно понятным и стандартизированным, что обеспечит удобство использования как для клиентов, так и для разработчиков, которые могут интегрировать приложение с другими сервисами.

**Основные требования к безопасности**

1. **Защита данных пользователей**. Поскольку приложение работает с персональными данными пользователей, такими как контактная информация и пароли, важно обеспечить надежную защиту этих данных. Пароли должны храниться в зашифрованном виде, используя безопасные алгоритмы хеширования (например, bcrypt или PBKDF2), чтобы предотвратить несанкционированный доступ к учетным записям. Для аутентификации рекомендуется использовать токены доступа, такие как JWT (JSON Web Token), что позволяет ограничить доступ к API для незарегистрированных пользователей.
2. **Защита от атак CSRF и XSS**. Приложение должно обеспечивать защиту от распространенных угроз безопасности, таких как межсайтовая подделка запросов (CSRF) и межсайтовый скриптинг (XSS). Защита от CSRF предотвращает попытки злоумышленников выполнить запрос от имени пользователя без его ведома, а защита от XSS ограничивает возможность внедрения вредоносных скриптов. В Django и Flask поддержка таких механизмов реализована встроенными инструментами, тогда как в FastAPI защита от CSRF может потребовать дополнительной настройки.
3. **Безопасность API и ограничение доступа**. API-приложения должен быть защищен от несанкционированного доступа. Это включает использование токенов для аутентификации пользователей и предоставление доступа к маршрутам приложения только авторизованным пользователям. Также важно реализовать механизмы ограничения количества запросов (rate limiting) для предотвращения атак, связанных с перегрузкой сервера. Это помогает не только защитить сервер от перегрузок, но и контролировать нагрузку при высоких объемах данных и запросов.
4. **Обработка и хранение аудиофайлов**. Приложение должно обеспечивать безопасное хранение аудиофайлов, загружаемых пользователями, а также генерируемых с помощью модели TTS. Важно ограничить доступ к этим файлам и защитить их от случайного удаления или несанкционированного скачивания. Загруженные файлы должны храниться в специально выделенных директориях на сервере и быть доступными только авторизованным пользователям через защищенные маршруты API. Это также включает возможность удаления файлов после обработки или в случае необходимости освободить место на сервере.
5. **Защита от SQL-инъекций и безопасное взаимодействие с базой данных**. Хотя современные фреймворки имеют встроенную защиту от SQL-инъекций, следует обеспечить правильное использование ORM и избегать прямых SQL-запросов, которые могут быть уязвимы для атак. В Django и FastAPI встроенные ORM предоставляют механизмы безопасного обращения к базе данных через классы и объекты Python, а Flask, при использовании SQLAlchemy, также поддерживает эту функциональность.

Основные требования к функционалу и безопасности приложения включают в себя не только разработку интуитивного и полезного интерфейса для выполнения задач классификации и генерации речи, но и строгие меры безопасности, обеспечивающие защиту данных и учетных записей пользователей. Учет данных требований позволит создать надежное и производительное приложение, устойчивое к типичным угрозам безопасности, что особенно важно для приложений, работающих с персональными и аудиоданными.

При реализации веб-приложения для классификации и генерации речи важно выбрать архитектуру, которая оптимально подходит для каждого из рассматриваемых фреймворков — Django, Flask и FastAPI. Каждый из этих фреймворков имеет свои уникальные особенности, подходы к обработке запросов, маршрутизации, работе с базой данных и масштабируемости. Подходящая архитектура для реализации функционала приложения с учетом этих факторов позволит максимально эффективно использовать возможности каждого фреймворка.

**Архитектура для Django**

Django — это фреймворк, построенный на основе архитектуры MVT (Model-View-Template), который предоставляет обширный набор встроенных инструментов для создания комплексных веб-приложений. Наилучшей архитектурой для Django будет разделение компонентов на слои модели, представления и шаблона. Основные элементы архитектуры для приложения на Django:

* **Модель (Model).** База данных и взаимодействие с ней реализуются через встроенную ORM Django. Все данные пользователя, такие как учетные записи, аутентификационные токены и пути к загруженным аудиофайлам, хранятся в базе данных и управляются с помощью моделей. Django ORM позволяет безопасно и удобно манипулировать данными, а также поддерживает миграции базы данных, что упрощает изменение структуры данных на стадии разработки.
* **Представления (View).** Представления обрабатывают запросы к приложению и управляют логикой маршрутов. В Django используется классический подход к представлениям, где каждый маршрут (например, загрузка файла, классификация или генерация аудио) реализован отдельной функцией или классом. Представления также взаимодействуют с моделями и шаблонами, обеспечивая обработку данных и передачу результатов пользователю.
* **Шаблоны (Template).** Для отображения информации пользователю Django использует систему шаблонов, которая позволяет динамически генерировать HTML-страницы. Шаблоны используются для отображения интерфейса загрузки аудиофайлов, отображения результатов классификации и предоставления ссылки на скачивание сгенерированного аудиофайла.

API. Django может использовать Django REST Framework (DRF) для создания REST API, что позволяет предоставить доступ к функционалу приложения через API-интерфейс. DRF обеспечивает удобную работу с сериализацией данных, поддерживает аутентификацию и управление разрешениями.

Django-архитектура подходит для проектов с комплексной бизнес-логикой, требующих встроенной безопасности, ORM и системы администрирования. Она лучше всего подходит для приложений, где важно удобство работы с данными, поддержка пользователей и централизованное управление.

**Архитектура для Flask**

Flask — это минималистичный фреймворк, который предоставляет разработчикам свободу выбора архитектурных решений. Оптимальной архитектурой для Flask будет модульная структура, которая облегчает добавление и замену компонентов. Основные архитектурные элементы для реализации приложения на Flask включают:

* **Модули для различных функций.** Flask позволяет создавать модули для каждой функции: загрузка файлов, классификация, генерация и аутентификация могут быть реализованы как отдельные модули, каждый из которых обрабатывает соответствующий маршрут. Это обеспечивает независимость компонентов, что упрощает тестирование и улучшает управляемость.
* **Расширения.** Flask требует дополнительных расширений для реализации функций, таких как работа с базой данных и аутентификацией. Для управления базой данных используется SQLAlchemy, что позволяет эффективно работать с SQL-запросами и сохранять пользовательские данные. Расширение Flask-Login отвечает за управление сессиями пользователей, что необходимо для безопасной аутентификации.
* **API.** В отличие от Django, Flask не имеет встроенного инструмента для API, однако создание маршрутов для API достаточно просто. В Flask каждый маршрут определяет действие, которое выполняется приложением, например, загрузка аудиофайла или генерация синтетической речи. Формат JSON используется для передачи данных между клиентом и сервером.
* **Шаблоны.** Flask также поддерживает использование шаблонов через Jinja2, что позволяет создавать динамические веб-страницы и управлять интерфейсом пользователя для каждой функции приложения.

Архитектура Flask является легковесной и гибкой, что делает её подходящей для небольших и средних приложений с требованием к модульности и простоте настройки. Она также удобна для разработки API и минимальных веб-интерфейсов, которые можно легко расширять по мере необходимости.

**Архитектура для FastAPI**

FastAPI — это асинхронный фреймворк, оптимизированный для создания высокопроизводительных API. Подходящей архитектурой для FastAPI будет асинхронная структура, ориентированная на быстрое выполнение запросов и параллельную обработку данных. Основные архитектурные элементы для реализации приложения на FastAPI включают:

* **Асинхронные маршруты.** FastAPI поддерживает асинхронные маршруты для всех основных функций приложения, таких как загрузка аудиофайлов, классификация и генерация речи. Асинхронная обработка позволяет эффективно распределять запросы и сокращать время отклика при работе с интенсивными задачами, такими как генерация и обработка аудио.
* **Модели и схемы данных.** FastAPI активно использует Pydantic для создания моделей данных и валидации запросов, что упрощает передачу данных между компонентами приложения и позволяет автоматически проверять типы данных. Это обеспечивает надежность при обработке JSON-данных, которые передаются через API.
* **Поддержка документации API.** FastAPI автоматически генерирует документацию API на основе аннотаций типов данных в маршрутах, что упрощает тестирование и делает приложение удобным для разработчиков, интегрирующих API. Сгенерированная документация позволяет пользователям видеть все доступные маршруты и параметры запросов, а также тестировать их прямо в браузере.
* **Интеграция с базой данных.** Для работы с базой данных FastAPI можно использовать SQLAlchemy или другие ORM-библиотеки. Асинхронная поддержка SQL-запросов позволяет эффективно взаимодействовать с базой данных, обеспечивая параллельную обработку нескольких запросов, что делает FastAPI идеальным выбором для приложений с высокими требованиями к производительности.
* **Система аутентификации.** Аутентификация пользователей в FastAPI может быть реализована с использованием JWT-токенов, что позволяет ограничить доступ к защищенным маршрутам и обеспечивать безопасность API.

Архитектура FastAPI наиболее подходит для высокопроизводительных, асинхронных приложений с поддержкой интенсивной обработки данных и сложных API. Это делает FastAPI идеальным выбором для создания API для задач классификации и генерации речи, где требуется минимизация задержек и поддержка параллельной обработки запросов.

Каждый из фреймворков — Django, Flask и FastAPI — имеет архитектуру, соответствующую их особенностям и задачам. Django лучше всего подходит для приложений, где важны комплексная бизнес-логика и поддержка пользователей. Flask предоставляет легковесную и гибкую архитектуру, удобную для модульных API и простых веб-интерфейсов. FastAPI, благодаря поддержке асинхронности, идеален для высоконагруженных API, требующих быстрой обработки данных в реальном времени.

## 2.2. Обучение модели классификации речи

Для обучения модели, способной различать настоящую и синтетическую речь, был подготовлен набор данных, состоящий из двух категорий: синтетической речи, сгенерированной с помощью модели текст-в-речь (TTS), и реальной речи, взятой из открытого источника. Подготовка данных была важным этапом, поскольку разнообразие и сбалансированность набора данных напрямую влияют на качество обучения и точность модели.

Для создания синтетической речи была использована модель Silero TTS, которая позволяет генерировать высококачественные аудиозаписи на русском языке. Модель Silero была выбрана за её высокую точность в передаче интонаций и естественности речи, что делало задачу классификации более сложной и, соответственно, интересной для модели машинного обучения. Модель версии v4\_ruс голосом Xenia была загружена с помощью библиотеки Torchи работала на процессоре (CPU), что не требовало дополнительных вычислительных мощностей.

Для генерации синтетической речи были подготовлены различные фразы на русском языке. Примеры фраз включали простые предложения, такие как "Привет, как у тебя дела сегодня?" и "Погода за окном выглядит замечательно". Эти фразы были выбраны, чтобы создать типичные примеры диалоговой речи, которые могли бы быть использованы в повседневном общении. Весь процесс генерации аудиофайлов происходил автоматически: текст передавался в модель, которая синтезировала аудиофайл, сохранённый в формате WAV с частотой дискретизации 48000 Гц. Каждый сгенерированный файл сохранялся в папку, предназначенную для синтетической речи, под уникальными именами, такими как "synthetic\_speech\_0.wav", "synthetic\_speech\_1.wav" и так далее. Это позволило создать набор из нескольких синтетических аудиофайлов, которые затем использовались для обучения модели. Код генерации речи приведен в листинге 1.

Листинг 1 – Код генерации речи

# Настройки для генерации

language = 'ru'

model\_id = 'v4\_ru' # Версия модели

sample\_rate = 48000 # Частота дискретизации

speaker = 'xenia' # Выбор говорящего

device = torch.device('cpu') # Используем CPU для генерации

# Загрузка модели Silero TTS для русского языка

model, example\_text = torch.hub.load(repo\_or\_dir='snakers4/silero-models',

model='silero\_tts',

language=language,

speaker=model\_id)

# Перенос модели на нужное устройство (в данном случае CPU)

model.to(device)

# Тексты для генерации

texts = [

"Привет, как у тебя дела сегодня?",

"Погода за окном выглядит замечательно."

]

# Генерация аудиофайлов

for i, text in enumerate(texts):

# Генерация аудио с помощью модели (передаем текст и говорящего)

audio = model.apply\_tts(text=text, speaker=speaker, sample\_rate=sample\_rate)

# Сохранение аудиофайла

file\_name = f"fake\_voice/synthetic\_speech\_{i}.wav" # Сохранение в формате WAV

torchaudio.save(file\_name, audio.unsqueeze(0), sample\_rate)

print(f"Файл {file\_name} успешно сохранен")

Для представления реальной речи был использован открытый набор данных с платформы Mozilla Common Voice, которая предоставляет записи голосов добровольцев на разных языках, включая русский. Этот набор данных был выбран по нескольким причинам: он содержит разнообразные аудиофайлы с записями речи разных людей, что добавляет в данные вариативность акцентов, интонаций и скоростей речи. Это было важно для того, чтобы модель научилась различать настоящую человеческую речь в самых разных её проявлениях.

Процесс подготовки данных настоящей речи включал загрузку и обработку аудиофайлов с платформы Common Voice. Все файлы были приведены к единому формату WAV (с помощью кода, представленного в листинге 2) с частотой дискретизации 48000 Гц для согласованности с синтетическими данными. Это обеспечило единообразие представленных данных, что упростило процесс обучения модели.

Листинг 2 – Код преобразования формата

# Функция для преобразования MP3 в WAV

def convert\_mp3\_to\_wav(mp3\_folder, wav\_folder):

# Проверяем наличие папки для сохранения файлов в формате WAV

if not os.path.exists(wav\_folder):

os.makedirs(wav\_folder)

# Проходим по всем файлам в папке

for file\_name in os.listdir(mp3\_folder):

if file\_name.endswith(".mp3"):

mp3\_path = os.path.join(mp3\_folder, file\_name)

wav\_path = os.path.join(wav\_folder, os.path.splitext(file\_name)[0] + ".wav")

# Загрузка MP3 файла

audio = AudioSegment.from\_mp3(mp3\_path)

# Конвертация в WAV и сохранение

audio.export(wav\_path, format="wav")

print(f"Файл {file\_name} успешно преобразован в WAV")

# Папка с файлами MP3

mp3\_folder = "check"

# Папка для сохраненных файлов WAV

wav\_folder = "real\_voice"

# Преобразование всех MP3 файлов в WAV

convert\_mp3\_to\_wav(mp3\_folder, wav\_folder)

Подготовленный набор данных включал в себя два класса: сгенерированную с помощью модели Silero TTS синтетическую речь и настоящую речь, взятую из открытого источника Mozilla Common Voice. Оба набора данных были структурированы и приведены к единому формату, что позволило создать сбалансированную и репрезентативную выборку для обучения модели. Такой подход обеспечил качественное различение двух типов речи, что является основой для успешной работы разработанной системы.

Для эффективного обучения модели классификации речи важным этапом является подготовка данных, что включает извлечение информативных признаков из аудиофайлов и создание обучающей выборки. В данном проекте для классификации настоящей и синтетической речи используется метод MFCC (Mel-Frequency Cepstral Coefficients), который позволяет представить аудиосигнал в форме числовых данных, максимально приближенных к восприятию звука человеческим ухом.

MFCC-признаки позволяют выделить главные частотные компоненты аудиосигнала, что особенно полезно для задачи классификации. Признаки извлекаются с использованием библиотеки librosa, которая предоставляет встроенные функции для обработки аудиофайлов. Основная функция, извлекающая MFCC, представлена в листинге 3. В данной функции extract\_mfcc происходит загрузка аудиофайла с помощью функции librosa.load, где файл автоматически конвертируется в массив значений звукового сигнала (y), а sr — это частота дискретизации. Параметр n\_mfcc задает количество коэффициентов MFCC для извлечения. Затем вычисленные MFCC-коэффициенты усредняются, чтобы получить одномерный вектор признаков, который будет использован для обучения модели.

Листинг 3 – Функция для извлечения признаков MFCC

import librosa

import numpy as np

# Функция для извлечения признаков MFCC из аудиофайла

def extract\_mfcc(file\_path, n\_mfcc=13):

y, sr = librosa.load(file\_path, sr=None)

mfcc = librosa.feature.mfcc(y=y, sr=sr, n\_mfcc=n\_mfcc)

return np.mean(mfcc, axis=1)

Для создания обучающих данных были подготовлены две папки, каждая из которых содержит набор аудиофайлов определенного типа:

REAL\_VOICE\_PATH — папка с записями настоящей речи.

FAKE\_VOICE\_PATH — папка с сгенерированной синтетической речью.

Каждый файл в этих папках обрабатывается функцией extract\_mfcc, которая извлекает признаки и сохраняет их вместе с метками классов. Метка 0 соответствует реальной речи, а метка 1 — синтетической. В листинге 4 представлен код для загрузки данных и извлечения MFCC. Этот код проходит по каждому аудиофайлу в заданных папках, проверяя, что он имеет формат .wav. Для каждого файла выполняется извлечение MFCC, и результат добавляется в массив признаков X, а соответствующая метка класса — в массив меток y.

Листинг 4 – Загрузка данных и извлечения MFCC

import os

# Пути к папкам с реальной и синтетической речью

REAL\_VOICE\_PATH = "real\_voice"

FAKE\_VOICE\_PATH = "fake\_voice"

# Инициализация массивов для признаков и меток

X = [] # признаки (MFCC)

y = [] # метки (0 - настоящая речь, 1 - синтетическая речь)

# Функция для загрузки и извлечения признаков из аудиофайлов

def load\_data(voice\_path, label):

for file\_name in os.listdir(voice\_path):

file\_path = os.path.join(voice\_path, file\_name)

if file\_path.endswith(".wav"): # только .wav файлы

mfcc = extract\_mfcc(file\_path)

X.append(mfcc)

y.append(label)

# Загрузка данных

load\_data(REAL\_VOICE\_PATH, 0) # метка 0 для настоящей речи

load\_data(FAKE\_VOICE\_PATH, 1) # метка 1 для синтетической речи

После загрузки и обработки всех аудиофайлов данные преобразуются в массивы numpy, что позволяет выполнять с ними операции, необходимые для обучения модели. Затем данные разделяются на обучающую и тестовую выборки с использованием train\_test\_split из библиотеки scikit-learn, что позволяет оценить точность модели на ранее невиданных данных. Разбиение происходит в пропорции 80:20, где 80% данных идут на обучение, а 20% — на тестирование. Код приведен в листинге 5.

Листинг 5 – Разбиение данных

from sklearn.model\_selection import train\_test\_split

import numpy as np

# Преобразование данных в формат numpy

X = np.array(X)

y = np.array(y)

# Разбиение данных на обучающую и тестовую выборки

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

Таким образом, весь процесс подготовки данных завершен, и модель теперь может использовать обучающую выборку для того, чтобы «научиться» различать реальную и синтетическую речь.

Для решения задачи классификации синтетической и реальной речи была разработана простая, но эффективная модель нейронной сети, способная различать эти два типа аудиосигналов на основе MFCC-признаков. Модель построена на базе библиотеки TensorFlow и включает несколько полносвязных (dense) слоев, обеспечивающих надежную классификацию на основе обучающих данных.

Модель разработана как последовательность полносвязных слоев, которая хорошо подходит для обработки низкоразмерных входных данных, таких как MFCC-признаки. В архитектуре используются полносвязные слои с функцией активации ReLU, которая помогает модели захватывать сложные зависимости в данных, а также слои Dropout для уменьшения риска переобучения и улучшения обобщающей способности.

Основная структура модели состоит из:

Входного слоя, принимающего вектор MFCC-признаков.

Трех скрытых слоев, состоящих из нейронов с активацией ReLU и слоев Dropout для повышения устойчивости к шуму.

Выходного слоя, содержащего один нейрон с сигмоидной активацией, который выдает вероятность принадлежности к одному из двух классов: реальной или синтетической речи.

Для создания модели используется библиотека TensorFlow и ее API Keras. В листинге 6 приведен код для построения модели.

Листинг 6 – Построение модели

import tensorflow as tf

from tensorflow.keras import layers

# Создание модели

def create\_model(input\_shape):

model = tf.keras.Sequential([

layers.InputLayer(input\_shape=input\_shape),

layers.Dense(256, activation='relu'), # Первый скрытый слой с 256 нейронами

layers.Dropout(0.3), # Dropout для предотвращения переобучения

layers.Dense(128, activation='relu'), # Второй скрытый слой с 128 нейронами

layers.Dropout(0.3),

layers.Dense(64, activation='relu'), # Третий скрытый слой с 64 нейронами

layers.Dense(1, activation='sigmoid') # Выходной слой для бинарной классификации

])

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

return model

После создания структуры модели необходимо обучить её на подготовленных данных. Для этого используется метод fit, который запускает процесс обучения с заданным количеством эпох. Каждая эпоха представляет собой полный цикл обучения на всех данных, после чего модель обновляет свои веса для улучшения классификационных способностей.

Перед началом обучения, данные были разделены на обучающую и тестовую выборки с целью проверки модели на невиданных данных. Процесс обучения начинается с инициализации модели и указания формы входных данных, которая соответствует количеству MFCC-признаков (см. листинг 7). В коде X\_train и y\_train — это обучающая выборка и метки классов для реальной и синтетической речи, а validation\_split=0.2 позволяет выделить 20% данных для валидации на каждой эпохе, что помогает отслеживать обобщающую способность модели в процессе обучения.

Листинг 7 – Обучение модели на обучающей выборке

# Инициализация модели с указанием формы входных данных

input\_shape = (X\_train.shape[1],) # Входная форма соответствует количеству признаков

model = create\_model(input\_shape)

# Обучение модели на обучающей выборке

history = model.fit(X\_train, y\_train, epochs=30, validation\_split=0.2)

После завершения обучения модель оценивается на тестовой выборке для получения метрик точности и качества классификации. Это позволяет понять, насколько хорошо модель справляется с классификацией на данных, которые она не видела ранее, что является важным этапом проверки ее производительности. Функция classification\_report (см. листинг 8) предоставляет подробные метрики, такие как точность, полнота и F1-оценка, для каждого класса, что позволяет комплексно оценить работу модели на тестовых данных. Округление предсказаний до 0 или 1 необходимо, так как выходной слой модели выдает вероятность принадлежности к классу, а не сам класс.

Листинг 8 – Вывод отчета о классификации

from sklearn.metrics import classification\_report

# Прогнозы модели на тестовой выборке

y\_pred = model.predict(X\_test)

y\_pred\_rounded = np.round(y\_pred) # Округляем предсказания до ближайшего целого (0 или 1)

# Вывод отчета о классификации

print(classification\_report(y\_test, y\_pred\_rounded))

Чтобы избежать повторного обучения при каждом запуске приложения, обученную модель можно сохранить в виде файла. Это позволяет в дальнейшем загружать модель и сразу использовать ее для классификации без необходимости заново обучать. Код сохранения представлен в листинге 9.

Листинг 9 – Сохранение обученной модели

# Сохранение обученной модели

model.save("speech\_classification\_model.h5")

Сохраненная модель может быть загружена в любое время с помощью функции load\_model, что удобно при интеграции модели в веб-приложение.

После сохранения модели ее можно использовать для классификации новых аудиофайлов. Для этого достаточно загрузить модель и использовать функцию predict для получения предсказаний. Пример функции, выполняющей классификацию нового аудиофайла, представлена в листинге 10. Функция predict\_voice принимает путь к аудиофайлу, извлекает MFCC-признаки и подает их в модель для классификации. Результат выводится в виде текста, указывающего, является ли файл настоящей или синтетической речью.

Листинг 10 – Сохранение обученной модели

# Загрузка обученной модели

trained\_model = tf.keras.models.load\_model("speech\_classification\_model.h5")

# Функция для предсказания на основе нового аудиофайла

def predict\_voice(file\_path, model):

mfcc = extract\_mfcc(file\_path)

mfcc = np.expand\_dims(mfcc, axis=0) # Преобразуем в форму (1, 13) для подачи в модель

prediction = model.predict(mfcc)

predicted\_label = np.round(prediction) # Округляем предсказание до 0 или 1

if predicted\_label == 0:

print(f"Файл {file\_path} классифицирован как настоящая речь.")

else:

print(f"Файл {file\_path} классифицирован как синтетическая речь.")

# Пример использования: классификация нового аудиофайла

new\_audio\_file = "test\_audio.wav" # Путь к новому аудиофайлу

predict\_voice(new\_audio\_file, trained\_model)

Созданная модель нейронной сети на основе полносвязных слоев показала хорошие результаты при классификации синтетической и реальной речи на основе MFCC-признаков. Архитектура модели и процесс обучения были адаптированы под специфику задачи, что позволяет применять эту модель для анализа и обработки аудиоданных в реальных приложениях.

После компиляции получим готовую модель speech\_classification\_model.h5, которая готова к интеграции в проект.

На рисунке 4 показано изменение потерь (ошибки) на обучающей и валидационной выборках в процессе обучения модели. По оси X отложены эпохи обучения, а по оси Y — значение функции потерь. Видно, что на начальных эпохах потери на обучающих данных (синяя линия) резко снижаются, что указывает на быстрое улучшение модели. Валидационные потери (оранжевая линия) также уменьшаются и стабилизируются на низком уровне, что говорит о хорошей способности модели обобщать данные. К концу обучения обе линии имеют низкие и близкие значения потерь, что указывает на отсутствие переобучения и хорошую общую производительность модели.

На рисунке 5 показана динамика точности модели на обучающей и валидационной выборках по мере обучения. По оси X указаны эпохи, а по оси Y — значение точности. Видно, что точность на валидационной выборке (оранжевая линия) быстро достигает уровня около 100% и остается стабильной на протяжении всего процесса обучения. Точность на обучающей выборке (синяя линия) постепенно повышается и стремится к 100%, но немного колеблется, что может указывать на небольшие изменения в процессе обучения. В целом, высокая и стабильная точность на обеих выборках свидетельствует о хорошей обобщающей способности модели.

На матрице ошибок (рисунок 6) показано распределение правильных и ошибочных предсказаний модели для задачи классификации реальной и синтетической речи. По диагонали видны корректные предсказания: 96 случаев реальной речи и 24 случая синтетической речи были правильно классифицированы. Отсутствие значений вне диагонали (нулевые значения) указывает на то, что модель не допустила ошибок и достигла идеальной точности на этих данных, верно определяя как реальные, так и синтетические записи. Такой результат демонстрирует высокую эффективность модели в классификации этих типов речи.

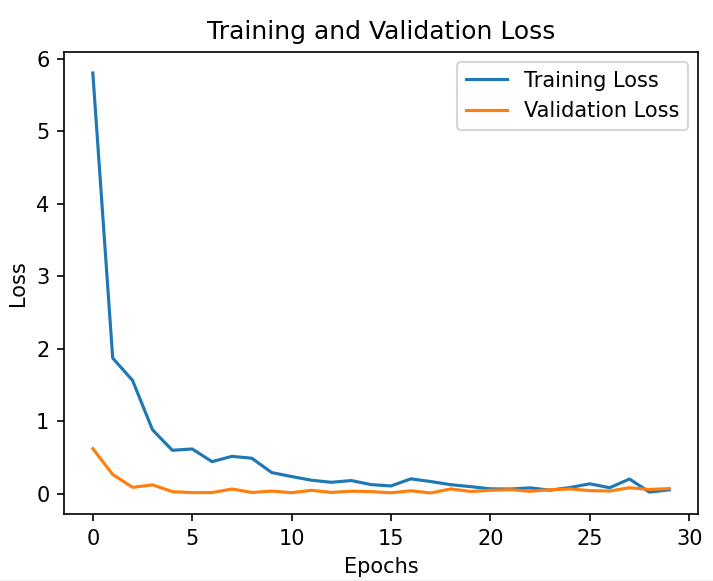


Рисунок 4 – График потерь

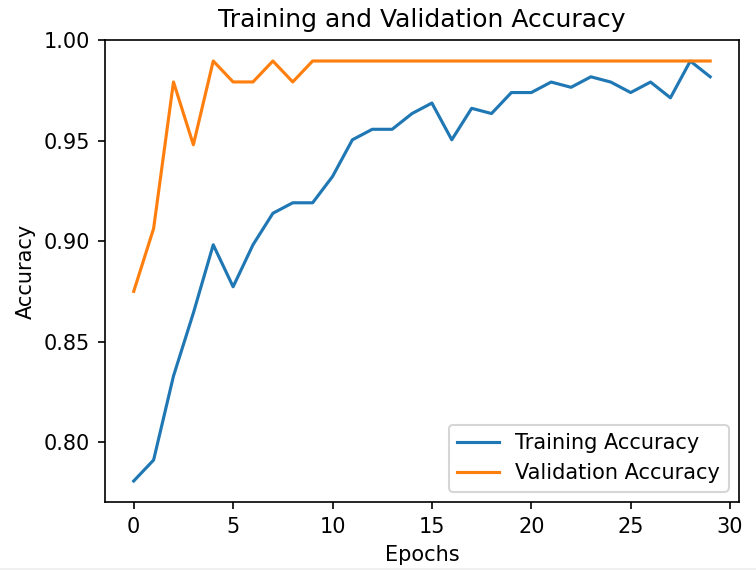


Рисунок 5 – График точности

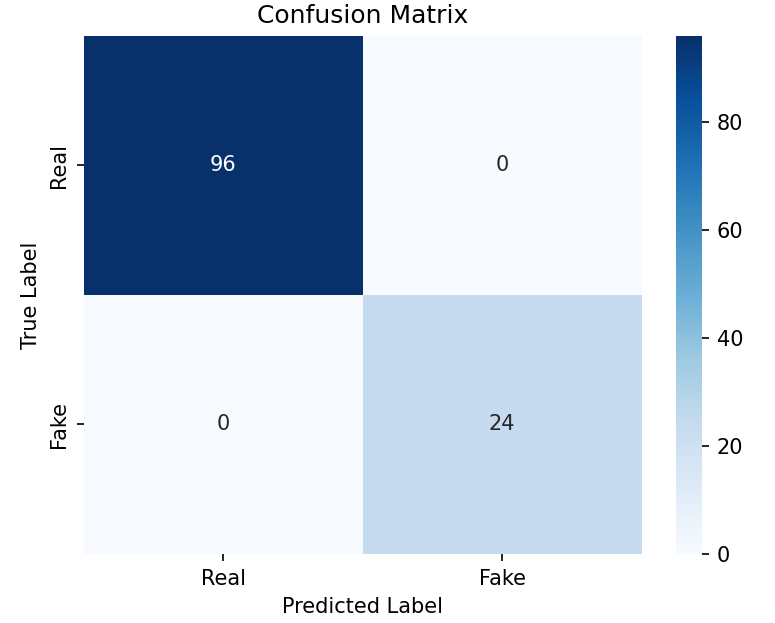


Рисунок 6 – Матрица ошибок

## 2.2. Проектирование компонентов приложения

**Регистрация и авторизация пользователей**

В FastAPI регистрация и авторизация будут реализованы с использованием номера телефона и пароля. В процессе регистрации данные пользователя (имя, фамилия, номер телефона и пароль) сохраняются в базе данных SQLite. Пароль предварительно хэшируется с использованием библиотеки bcrypt, что защищает его в случае утечки данных. При регистрации приложение проверяет уникальность номера телефона; если номер уже зарегистрирован, пользователю отображается ошибка.

Для авторизации в FastAPI планируется использование токена JWT (JSON Web Token), который создается при успешном входе пользователя и передается в cookie. При каждом запросе к защищенным маршрутам приложение проверяет наличие и валидность токена, предоставляя доступ к функционалу только аутентифицированным пользователям. Пользователь также может выйти из системы, удалив cookie с токеном, что завершает сеанс.

В Django регистрация и авторизация будут строиться на основе данных о пользователе, таких как имя, фамилия, телефон и пароль, но в этом случае используется встроенная система аутентификации Django. Пользователи регистрируются через форму, и их данные, включая хэшированный пароль, сохраняются в базе данных. Номер телефона используется в качестве имени пользователя (username), что позволяет упростить процесс аутентификации.

Для входа в систему Django предоставляет метод authenticate, который проверяет введенные данные. Если данные верны, пользователь входит в систему и получает доступ к защищенным маршрутам, таким как классификация и генерация речи. Благодаря встроенному декоратору @login\_required, приложение ограничивает доступ к этим маршрутам для неавторизованных пользователей, перенаправляя их на страницу логина. Пользователь также может выйти из системы, вызвав метод logout, который завершает текущий сеанс.

В Flask регистрация и авторизация пользователей будут построены с использованием SQLite и сессий. Пользователь регистрируется, вводя имя, фамилию, номер телефона и пароль. Приложение хэширует пароль с помощью werkzeug.security.generate\_password\_hash, прежде чем сохранить его в базе данных. Во время регистрации Flask проверяет уникальность номера телефона и предотвращает регистрацию дублирующихся учетных записей.

Для авторизации Flask использует сессионный подход: при успешном входе данные о пользователе сохраняются в сессии, что позволяет сохранять его аутентификацию между запросами. Чтобы ограничить доступ к защищенным маршрутам, таким как классификация и генерация речи, используется декоратор @login\_required. Он проверяет наличие пользователя в сессии и перенаправляет на страницу логина, если пользователь не авторизован. Выход из системы реализован через очистку сессии, что завершает авторизацию.

**Классификация речи и генерация аудиофайлов**

В проекте будут реализованы процессы классификации речи и генерации аудиофайлов, которые позволят системе различать настоящую и синтетическую речь, а также преобразовывать текст в аудио. Эти функции будут обеспечены использованием современных методов машинного обучения и синтеза речи, что позволит создать приложение с возможностью интерактивной работы с аудиофайлами.

Классификация речи будет направлена на автоматическое различение записей реальной и синтетической речи. Для этого сначала будет осуществлено извлечение признаков MFCC (Mel-frequency cepstral coefficients), которые представляют спектральные особенности аудиосигнала. MFCC будут характеризовать распределение частотных составляющих звука, что позволит модели "понимать" различия между синтетическим и настоящим звуком. Для извлечения этих признаков будет использована библиотека `librosa`, которая обработает загружаемый аудиофайл и преобразует его в коэффициенты MFCC. Эти данные будут поданы на вход модели классификации — нейронной сети, обученной на наборе данных, содержащем примеры настоящей и синтетической речи. Во время обучения модель будет настраиваться так, чтобы минимизировать ошибку и правильно классифицировать звуки. После завершения обучения модель будет применяться для анализа новых аудиофайлов: при загрузке файла система обработает его и подаст полученные MFCC на вход модели, которая выдаст метку, указывающую на то, является ли речь настоящей или синтетической. Это позволит пользователям получать автоматические результаты анализа для любых загруженных аудиофайлов.

Генерация речи, или синтез аудио на основе текста, будет основана на модели TTS (Text-to-Speech), которая позволит преобразовать текст в звучащую речь. Для выполнения этой задачи будет использоваться модель из библиотеки Silero, адаптированная для русского языка. Модель примет текст, вводимый пользователем, и синтезирует его в звуковой форме, создавая аудиофайл с естественным голосом, подходящий для различных задач. Пользователь сможет выбрать параметры, такие как голос и частота дискретизации, что позволит регулировать звучание результата. Созданный аудиофайл будет сохраняться на сервере и станет доступен для загрузки или прослушивания, предоставляя пользователю возможность использовать результат для личных или профессиональных нужд.

Интеграция этих функций будет осуществлена в приложении с использованием популярных веб-фреймворков, таких как FastAPI, Django и Flask, что обеспечит удобный доступ к функциям классификации и генерации. В FastAPI классификация речи и генерация синтетического аудио будут реализованы через маршруты, защищенные аутентификацией; пользователь получит доступ к этим функциям после входа в систему. В Django также будут использоваться защищенные представления, а встроенная система хранения файлов поможет управлять загружаемыми и генерируемыми файлами. В Flask приложение будет поддерживать сессии для управления авторизацией, а загружаемые файлы будут обрабатываться с использованием простых маршрутов. Это решение позволит эффективно и безопасно интегрировать функции классификации и генерации в приложение, предлагая пользователям простой интерфейс для работы с аудиофайлами.

**Защита маршрутов и обеспечение безопасности данных**

Для защиты маршрутов и обеспечения безопасности данных в приложении будут использоваться аутентификация и шифрование данных. Маршруты, связанные с классификацией и генерацией речи, будут защищены с помощью аутентификации, доступ к ним получат только авторизованные пользователи. В FastAPI и Django будет применяться JWT (JSON Web Token) или сессионное управление для контроля доступа, а в Flask — сессионные данные, защищенные секретным ключом приложения.

Пароли пользователей будут хэшироваться перед сохранением в базе данных с помощью bcrypt (FastAPI, Django) или werkzeug.security (Flask), что предотвратит утечку паролей в случае взлома базы данных. Все соединения будут происходить через HTTPS, обеспечивая шифрование данных, передаваемых между клиентом и сервером, что защитит информацию от перехвата.

# ГЛАВА 3. РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ

## 3.1. Реализация с использованием Django

В рамках реализации модуля аутентификации и базы данных в Django используется встроенная модель User, которая позволяет управлять пользователями и обеспечивать безопасное хранение данных. Django предоставляет встроенные механизмы для хранения паролей в хэшированном виде и защиты от атак на аккаунты. В данном случае в качестве уникального идентификатора пользователя используется номер телефона, а данные, такие как имя, фамилия, телефон и пароль, сохраняются в базе данных SQLite.

Настройка базы данных выполняется в конфигурации проекта в файле settings.py, где указывается тип базы данных и путь к файлу SQLite. Django автоматически создает необходимые таблицы, включая таблицу пользователей, которая будет хранить данные, используемые для аутентификации [11].

Для регистрации пользователей разработано представление register\_view, которое обрабатывает форму, отправленную через шаблон registration.html. Пользователь заполняет поля имени, фамилии, номера телефона и пароля. Представление проверяет уникальность номера телефона: если пользователь с таким номером уже существует, выводится сообщение об ошибке, предотвращающее регистрацию дублирующихся учетных записей. После успешной регистрации пользователь автоматически авторизуется и получает доступ к основным функциям приложения. Код представлен в листинге 11. Для авторизации пользователей используется представление login\_view, которое обрабатывает вводимые номер телефона и пароль. В случае успешной авторизации пользователь получает доступ к защищенным маршрутам, таким как классификация и генерация речи. В случае неудачной попытки входа отображается сообщение об ошибке.

Листинг 10 – Представление регистрации

# Представление регистрации

def register\_view(request):

if request.method == 'POST':

phone = request.POST.get('phone')

password = request.POST.get('password')

if User.objects.filter(username=phone).exists():

return render(request, 'registration.html', {'error': 'Такой пользователь уже зарегистрирован.'})

user = User.objects.create\_user(username=phone, password=password)

login(request, user)

return redirect('classify\_speech')

return render(request, 'registration.html')

Django также позволяет ограничить доступ к определенным маршрутам с помощью декоратора @login\_required, который автоматически перенаправляет неавторизованных пользователей на страницу входа. Это обеспечивает дополнительную защиту и контроль за доступом к важным функциям, таким как классификация и генерация речи, которые требуют наличия авторизованного пользователя.

Таким образом, интеграция системы аутентификации в Django предоставляет полноценный механизм защиты данных, включая хэширование паролей, проверку уникальности данных при регистрации и защиту маршрутов для авторизованных пользователей. Все компоненты, включая представления, шаблоны и маршруты, создают надежную систему управления пользователями, обеспечивая при этом удобный и безопасный доступ к функционалу приложения.

В приложении реализован API для классификации речи и генерации аудиофайлов, обеспечивающий взаимодействие с клиентом по HTTP-запросам. Этот API позволяет загружать аудиофайлы для классификации и отправлять текст для генерации синтетического аудио. Использование таких методов дает возможность интеграции приложения с другими сервисами, а также облегчает взаимодействие с фронтенд-частью через AJAX или REST-запросы.

API для классификации речи позволяет пользователю загрузить аудиофайл, после чего система обрабатывает его, извлекая признаки MFCC (Mel-frequency cepstral coefficients) для анализа. Эти признаки затем подаются на вход модели машинного обучения, которая возвращает результат классификации — реальная или синтетическая речь. В случае успешного анализа система возвращает JSON-ответ с результатом классификации и ссылкой на аудиофайл, что удобно для фронтенд-отображения. Код реализованной функциональности представлен в листинге 11. В коде classify\_speech\_api принимает POST-запрос с загруженным файлом, после чего сохраняет его на сервере, извлекает признаки MFCC и подает их на вход модели. По завершении классификации возвращается JSON-ответ с результатом. Декоратор @login\_required защищает маршрут, ограничивая доступ к API только для авторизованных пользователей.

Листинг 11 – Код классификации

# views.py

from django.http import JsonResponse

from django.contrib.auth.decorators import login\_required

@login\_required(login\_url='login')

def classify\_speech\_api(request):

if request.method == 'POST' and request.FILES.get('file'):

audio\_file = request.FILES['file']

file\_path = save\_file(audio\_file)

mfcc = extract\_mfcc(file\_path)

prediction = classification\_model.predict(mfcc)

result = "Настоящая речь" if np.round(prediction)[0][0] == 0 else "Синтетическая речь"

return JsonResponse({'result': result, 'audio\_url': file\_path})

return JsonResponse({'error': 'Файл не загружен или неверный запрос'}, status=400)

Для генерации речи создан API, который принимает текстовую строку и преобразует ее в аудио с помощью модели Text-to-Speech. Модель TTS используется для создания звукового файла, который затем сохраняется на сервере. В случае успешного выполнения запроса API возвращает JSON-ответ с URL на сгенерированный аудиофайл, который может быть загружен или воспроизведен на клиентской стороне. В листинге 12 показан код, в нем generate\_audio\_api принимает текст от пользователя, передает его в модель TTS, которая генерирует аудио. Сгенерированный файл сохраняется, и возвращается ссылка на его загрузку. Использование csrf\_exempt снимает защиту от CSRF-атак для этого маршрута, что полезно для взаимодействия с внешними клиентами, например, при интеграции с другими приложениями.

Листинг 12 – Генерация аудио

# views.py

from django.views.decorators.csrf import csrf\_exempt

@login\_required(login\_url='login')

@csrf\_exempt

def generate\_audio\_api(request):

if request.method == 'POST':

text = request.POST.get('text', '')

if not text:

return JsonResponse({'error': 'Текст не может быть пустым'}, status=400)

audio\_path = generate\_audio\_file(text)

return JsonResponse({'audio\_url': audio\_path})

return JsonResponse({'error': 'Неверный метод запроса'}, status=400)

Оба API маршрута обеспечивают удобный и безопасный способ работы с функционалом классификации и генерации речи, позволяя клиентам загружать аудиофайлы, получать классификацию, а также преобразовывать текст в синтетическую речь.

При запуске приложения видим форму авторизации (см. рисунок 7).

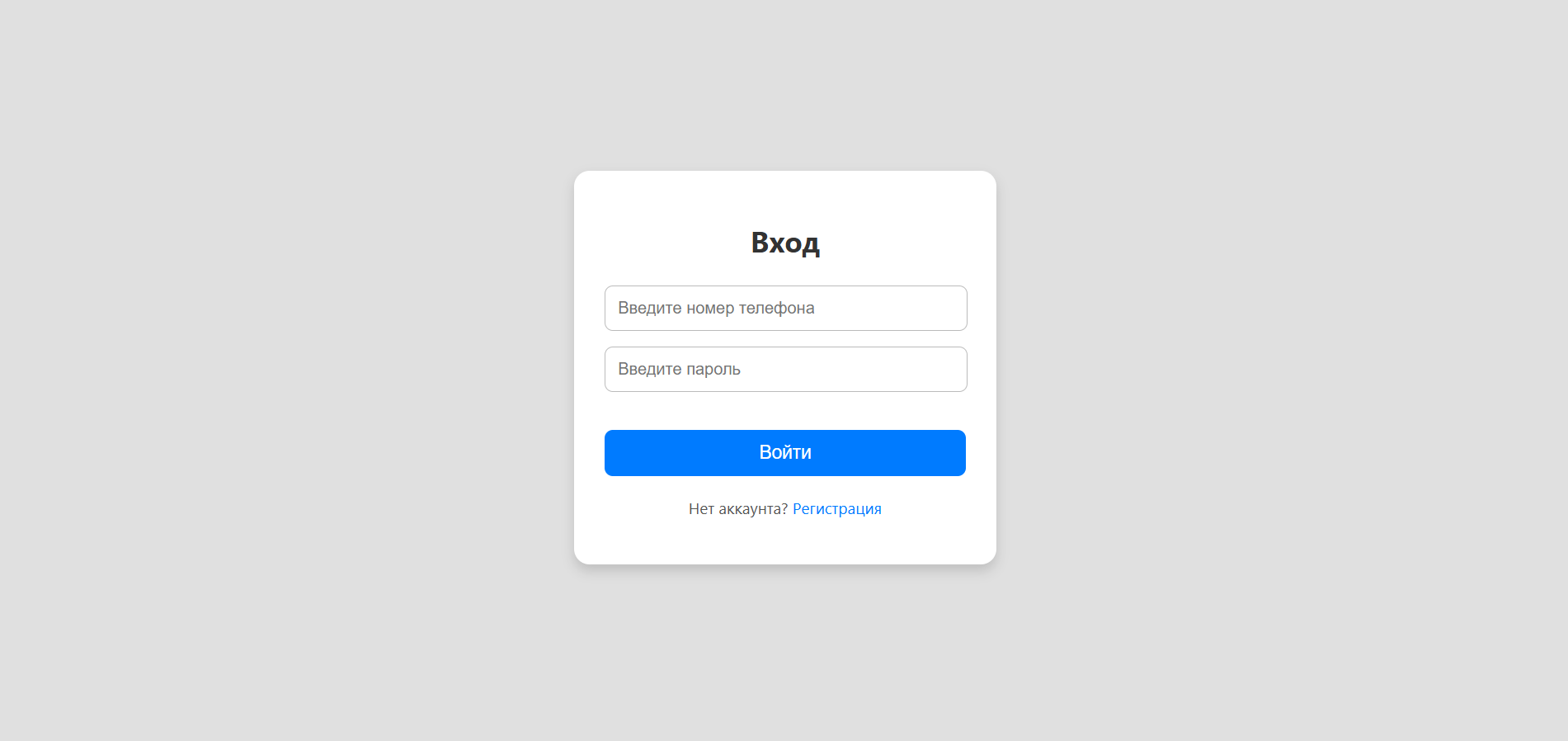


Рисунок 7 – Авторизация

При нажатии по ссылке «Регистрация» перейдем на страницу регистрации, страница представлена на рисунке 8.

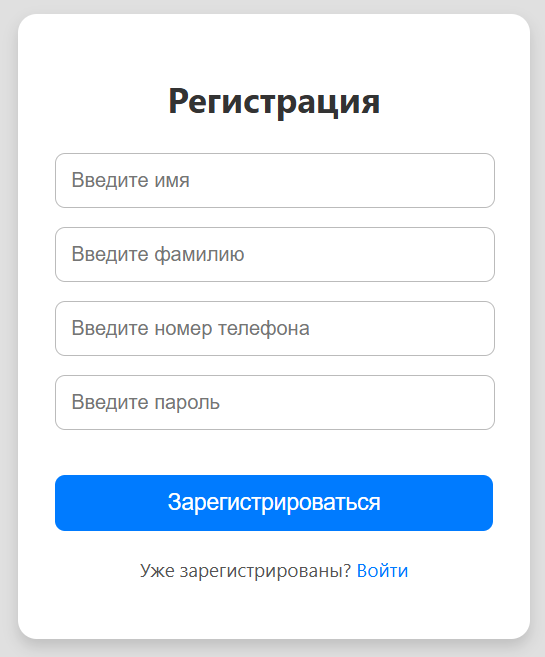


Рисунок 8 – Регистрация

После успешной регистрации и авторизации перейдем на главную страницу приложения, на которой можно классифицировать аудио файл (см. рисунок 9).

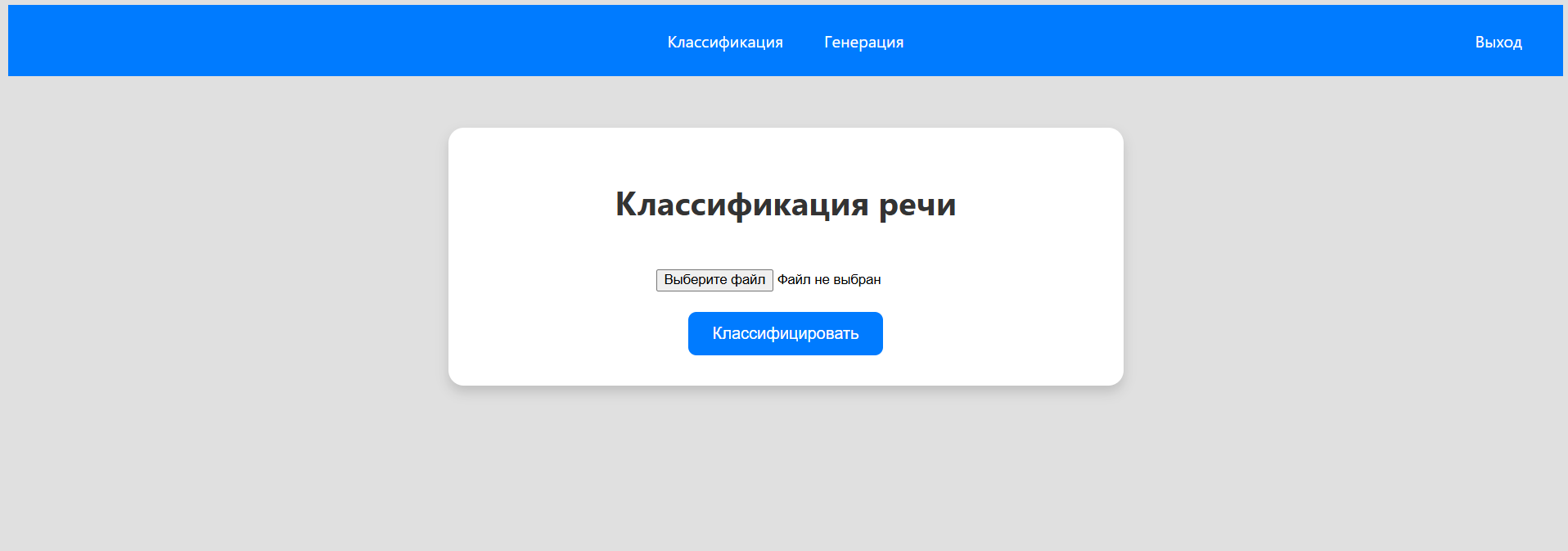


Рисунок 9 – Страница классификации

При выборе файла и нажатии кнопки «Классифицировать» система предоставит результат классификации и можно будет послушать выбранный файл, как показано на рисунке 10.

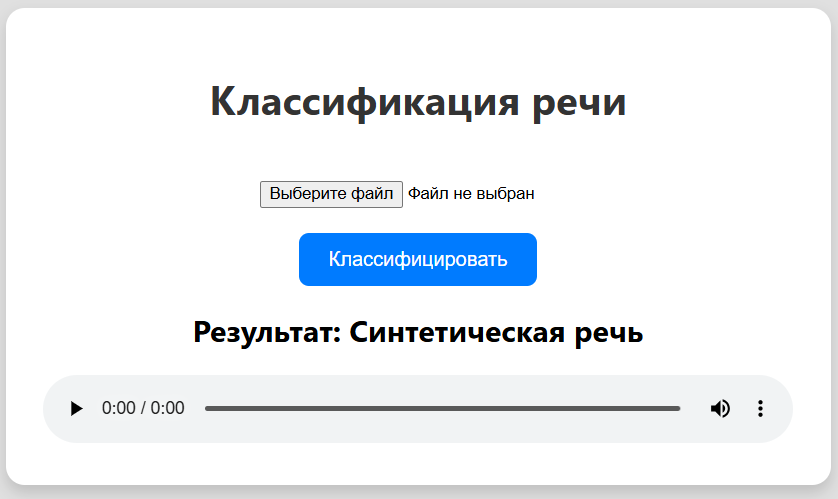


Рисунок 10 – Результат классификации

При переходе на вкладку «Генерация» увидим текстовое поле для ввода (см. рисунок 11).

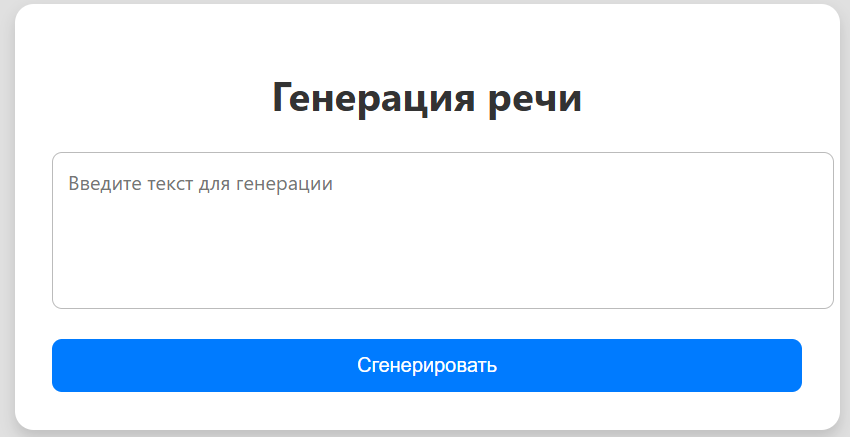


Рисунок 11 – Страница генерации текста

После нажатия на кнопку «Сгенерировать» сможем прослушать сгенерированный аудио файл и скачать его. Страница представлена на рисунке 11.

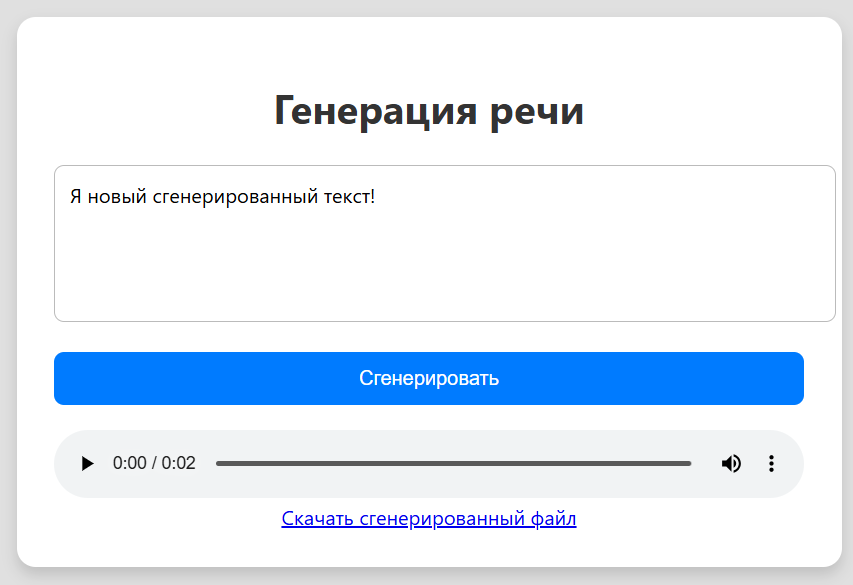


Рисунок 12 – Текст преобразован в речь

Полный код Django системы представлен в приложении А.

## 3.2. Разработка на Flask

Flask является легковесным веб-фреймворком, что позволяет быстро и эффективно создавать API для различных целей, включая классификацию и генерацию речи. В данном приложении реализован набор маршрутов, предоставляющих доступ к функционалу классификации загруженных аудиофайлов и генерации синтетической речи. Такая структура делает приложение гибким и расширяемым, предоставляя разработчикам возможность легко добавлять или изменять функциональные элементы.

Flask ориентирован на маршрутизацию HTTP-запросов, которая позволяет создавать упрощенные API-эндпоинты для каждой функции. Основной частью системы является API для классификации загруженных аудиофайлов, где пользователь может отправить файл в формате .wav, после чего система обработает его и вернет результат классификации. Маршрут для классификации (/upload) принимает POST-запрос с аудиофайлом, который проходит через предобработку с выделением признаков MFCC (Mel-frequency cepstral coefficients), необходимых для анализа. Модель машинного обучения на основе нейронной сети обрабатывает эти признаки и выдает предсказание — реальная или синтетическая речь [8].

Маршрут, представленный в листинге 13 защищен декоратором @login\_required, что гарантирует доступ к нему только авторизованным пользователям. После загрузки аудиофайл обрабатывается, и результат передается обратно в шаблон classification.html, где он отображается вместе с ссылкой для загрузки.

Листинг 13 – Маршрут для классификации аудиофайла

# Маршрут для классификации аудиофайла

@app.route('/upload', methods=['POST'])

@login\_required

def upload\_file():

if 'file' not in request.files:

return redirect(request.url)

file = request.files['file']

if file and file.filename.endswith('.wav'):

file\_path = os.path.join('uploads', file.filename)

file.save(file\_path)

# Извлечение MFCC и классификация

y, sr = librosa.load(file\_path, sr=None)

mfcc = np.mean(librosa.feature.mfcc(y=y, sr=sr, n\_mfcc=13), axis=1).reshape(1, -1)

prediction = classification\_model.predict(mfcc)

result = "Настоящая речь" if np.round(prediction)[0][0] == 0 else "Синтетическая речь"

return render\_template('classification.html', result=result, audio\_file=file.filename)

return redirect(url\_for('classification'))

В приложении также реализован маршрут для генерации речи (/generate\_audio), который принимает текст от пользователя, передает его в модель TTS (Text-to-Speech), а затем сохраняет сгенерированный аудиофайл на сервере. Пользователь получает ссылку на скачивание сгенерированного файла, что делает процесс получения синтетической речи простым и доступным. Этот код представлен в листинге 14. Эта реализация API позволяет пользователю отправлять текстовые данные через POST-запрос, после чего TTS-модель генерирует аудиофайл. Готовый файл сохраняется в каталоге сервера и становится доступным для скачивания. Путь к файлу возвращается в формате JSON, что позволяет легко интегрировать функционал с другими частями приложения или внешними сервисами.

Листинг 14 – Маршрут для классификации аудиофайла

# Маршрут для генерации аудио на основе текста

@app.route('/generate\_audio', methods=['POST'])

@login\_required

def generate\_audio():

text = request.form.get('text')

if not text:

return redirect(url\_for('generation'))

audio = generation\_model.apply\_tts(text=text, speaker=speaker, sample\_rate=sample\_rate)

file\_path = os.path.join('generated\_audio', "synthetic\_speech.wav")

torchaudio.save(file\_path, audio.unsqueeze(0), sample\_rate)

return {'audio\_url': url\_for('download\_audio', filename="synthetic\_speech.wav")}

Flask обеспечивает гибкую и легковесную архитектуру, в которой маршрутизация позволяет легко управлять API для классификации и генерации речи, защищая их с помощью сессионного управления.

Функциональность генерации и классификации речи позволяет пользователям загружать аудиофайлы для анализа и получать синтезированный аудиоответ на основе текстового ввода. Эти функции требуют интеграции двух отдельных моделей: одна используется для классификации (распознавания реальной или синтетической речи), а другая — для генерации речи (TTS). Flask обеспечивает легкость реализации этих функций через маршрутизацию и минималистичный API.

Для классификации речи реализован маршрут, который принимает аудиофайл, извлекает его признаки и отправляет на анализ модели машинного обучения. Аудиофайл загружается пользователем в формате .wav, и для обработки используется библиотека librosa, которая выделяет признаки MFCC (коэффициенты мел-частотных кепстральных признаков). Эти признаки подаются на вход модели, которая затем возвращает предсказание — настоящая или синтетическая речь. Код представлен в листинге 15. Маршрут /upload обрабатывает POST-запрос и позволяет пользователю загрузить файл. После загрузки файл сохраняется, и librosa выделяет MFCC-признаки [5 ], которые используются для предсказания модели. Результат классификации возвращается в шаблон classification.html для отображения пользователю. Использование @login\_required ограничивает доступ к этому маршруту, что предотвращает несанкционированное использование ресурсоемкой функции.

Листинг 15 – Код классификации

@app.route('/upload', methods=['POST'])

@login\_required

def upload\_file():

file = request.files.get('file')

if not file or not file.filename.endswith('.wav'):

flash('Пожалуйста, загрузите файл формата .wav')

return redirect(url\_for('classification'))

# Сохранение файла и извлечение признаков

file\_path = os.path.join('uploads', file.filename)

file.save(file\_path)

y, sr = librosa.load(file\_path, sr=None)

mfcc = np.mean(librosa.feature.mfcc(y=y, sr=sr, n\_mfcc=13), axis=1).reshape(1, -1)

# Классификация с использованием модели

prediction = classification\_model.predict(mfcc)

result = "Настоящая речь" if np.round(prediction)[0][0] == 0 else "Синтетическая речь"

return render\_template('classification.html', result=result, audio\_file=file.filename)

Для синтеза речи на основе текстового ввода реализован маршрут /generate\_audio (см. листинг 16), который принимает текстовые данные, генерирует аудиофайл и возвращает пользователю ссылку для его скачивания. Генерация речи выполняется с помощью TTS-модели Silero, которая позволяет синтезировать качественную русскоязычную речь. Пользователь вводит текст, и модель возвращает аудиофайл, который сохраняется на сервере и становится доступным для скачивания. Этот маршрут получает текст от пользователя и передает его в TTS-модель, которая генерирует аудио на основе введенного текста. Сгенерированный файл сохраняется в каталоге generated\_audio, и возвращается URL для скачивания. Подход с сохранением файла на сервере позволяет пользователю не только прослушивать аудио в браузере, но и загружать его для дальнейшего использования.

Листинг 16 – Маршрут генерации

@app.route('/generate\_audio', methods=['POST'])

@login\_required

def generate\_audio():

text = request.form.get('text')

if not text:

flash('Поле текста не может быть пустым')

return redirect(url\_for('generation'))

# Генерация аудиофайла

audio = generation\_model.apply\_tts(text=text, speaker=speaker, sample\_rate=sample\_rate)

file\_name = f"generated\_{hash(text)}.wav"

file\_path = os.path.join('generated\_audio', file\_name)

torchaudio.save(file\_path, audio.unsqueeze(0), sample\_rate)

return {'audio\_url': url\_for('download\_audio', filename=file\_name)}

Функциональность генерации и классификации речи в Flask легко реализуется за счет гибкости маршрутизации и минималистичной структуры API. Пользователи могут быстро и просто загружать аудио для анализа или получать сгенерированный текст, что делает приложение удобным и универсальным для различных задач, связанных с обработкой речи.

При запуске системы видим страницу авторизации, представленную на рисунке 13. Страница регистрации представлена на рисунке 14. После успешной авторизации попадаем на страницу классификации, представленную на рисунке 15, при нажатии на кнопку «Генерация» видим страницу, показанную на рисунке 16. Функционал приложения полностью соответствует приложению на Django.

Полный код Flask системы представлен в приложении Б.

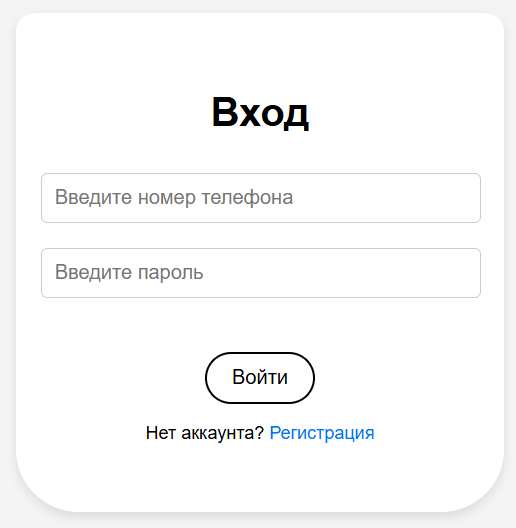


Рисунок 13 – Вход

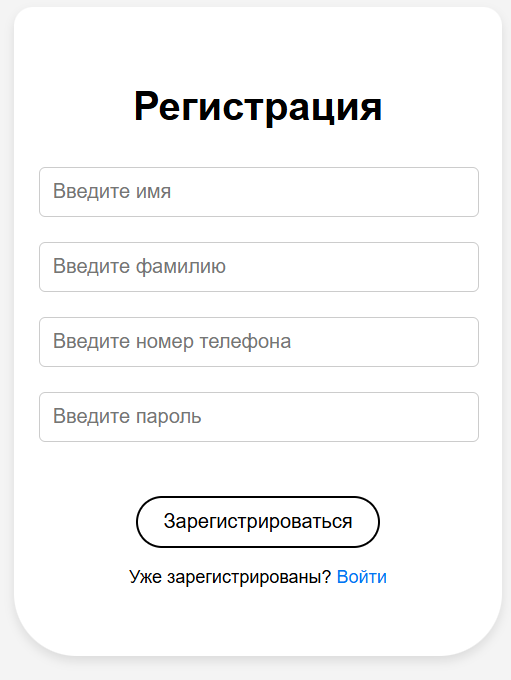


Рисунок 14 – Регистрация

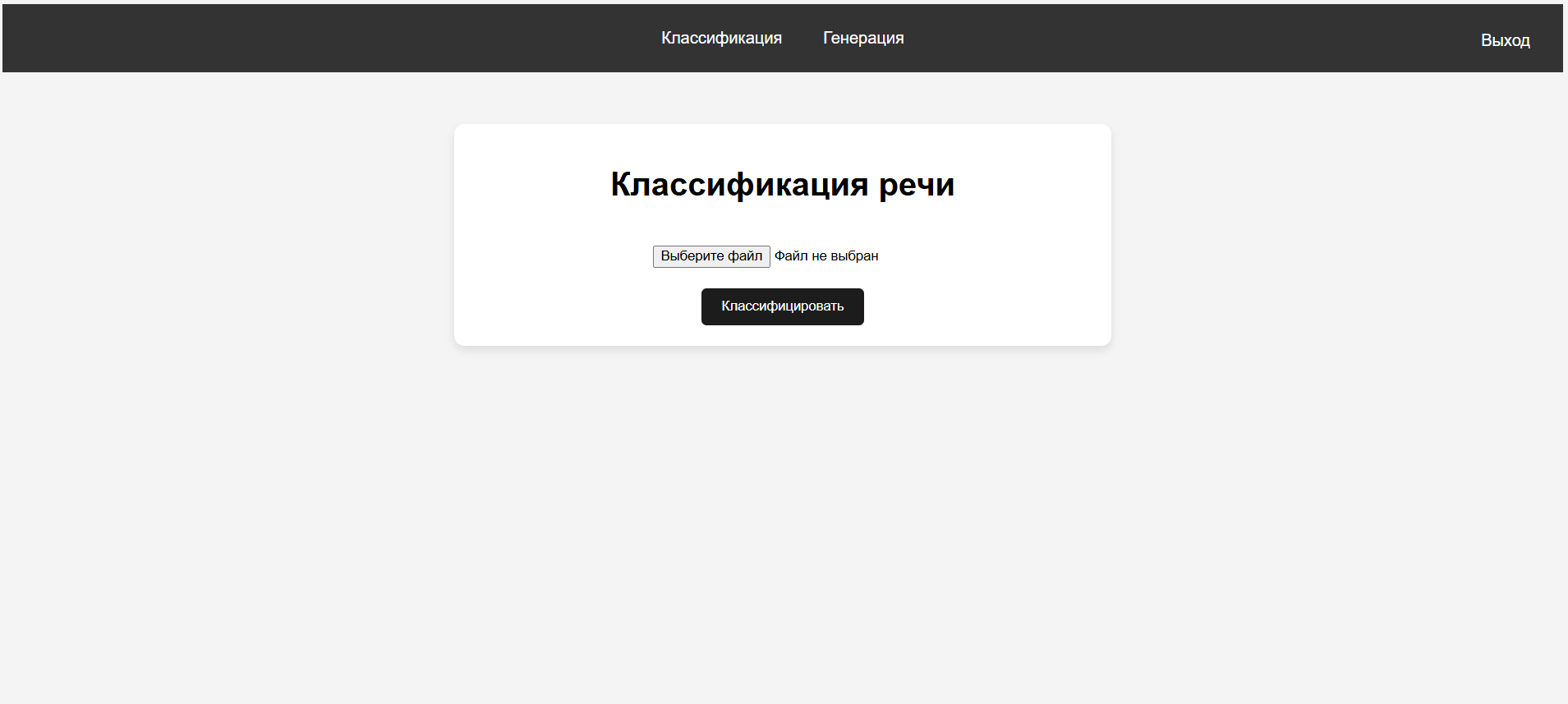


Рисунок 15 – Классификация

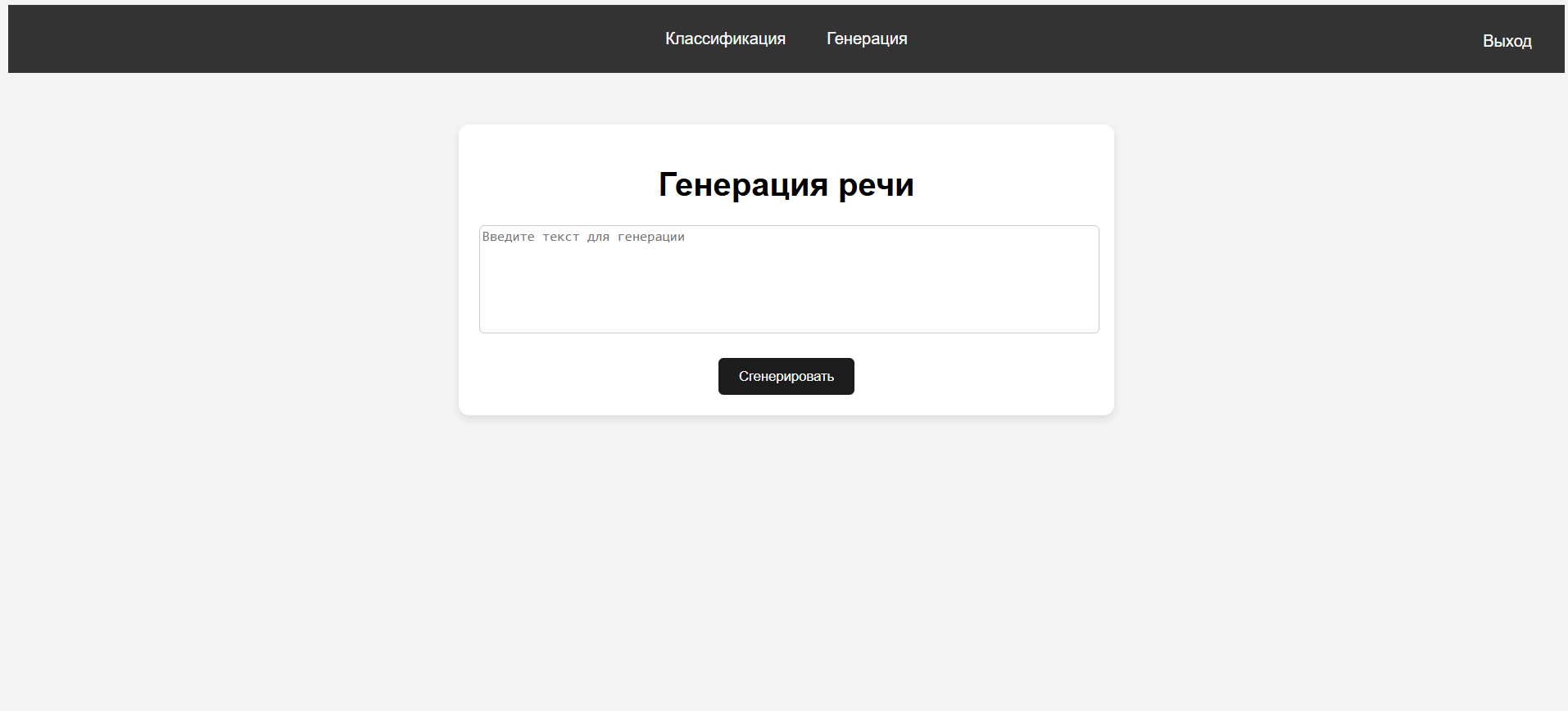


Рисунок 16 – Генерация

## 3.3. Разработка с использованием FastAPI

FastAPI, с его асинхронной архитектурой, предоставляет идеальные условия для оптимизации и масштабирования API, особенно для ресурсоемких задач, таких как классификация и генерация речи. Асинхронное взаимодействие позволяет серверу обрабатывать запросы более эффективно, поддерживая одновременно несколько подключений, что особенно важно для обработки аудио- и текстовых данных в реальном времени.

Асинхронное API для классификации речи принимает загружаемый пользователем аудиофайл, обрабатывает его для извлечения характеристик (признаков MFCC) и подает их на вход обученной модели для классификации — определяя, является ли речь настоящей или синтетической. Функция принимает UploadFile в качестве входного параметра, позволяя асинхронно загружать файл на сервер. Это помогает избежать блокировки других запросов, так как загрузка и обработка файлов в FastAPI происходит параллельно. В листинге 17 функция upload\_file выполняет асинхронное чтение файла через await file.read(), затем передает его на обработку [9]. Это позволяет серверу сразу отвечать на другие запросы, не дожидаясь завершения работы функции. Выделение признаков MFCC и предсказание производятся с использованием модели машинного обучения, а результат передается обратно пользователю в шаблон index.html.

Листинг 17 – Извлечение MFCC

@app.post("/upload", response\_class=HTMLResponse)

async def upload\_file(

request: Request, file: UploadFile = File(...), current\_user: User = Depends(get\_current\_user)

):

file\_location = os.path.join(UPLOAD\_FOLDER, file.filename)

with open(file\_location, "wb") as f:

f.write(await file.read())

mfcc = extract\_mfcc(file\_location)

mfcc = np.expand\_dims(mfcc, axis=0)

prediction = classification\_model.predict(mfcc)

result = "Настоящая речь" if np.round(prediction)[0][0] == 0 else "Синтетическая речь"

return templates.TemplateResponse("index.html", {

"request": request,

"result": result,

"audio\_file": file.filename

})

Маршрут для генерации речи также асинхронный и поддерживает обработку нескольких запросов одновременно, что повышает производительность при работе с текстовыми запросами для синтеза речи. Пользователь отправляет текст, который асинхронно передается в модель TTS (Text-to-Speech) для генерации речи. Модель создает аудиофайл, который сохраняется на сервере и становится доступным для скачивания. В листинге 18 асинхронный метод generate\_audio генерирует аудио из текста с использованием модели TTS и сохраняет его на сервере. URL-адрес сгенерированного файла возвращается в JSON-ответе, что позволяет пользователю загрузить аудиофайл или воспроизвести его непосредственно в браузере. Асинхронная архитектура маршрута позволяет системе генерировать аудио для нескольких пользователей одновременно, не задерживая ответы по другим запросам.

Листинг 18 – Маршрут генерации речи

@app.post("/generate\_audio", response\_class=JSONResponse)

async def generate\_audio(

text: str = Form(...), current\_user: User = Depends(get\_current\_user)

):

if not text:

return JSONResponse({"error": "Текст для генерации пуст"}, status\_code=400)

audio = tts\_model.apply\_tts(text=text, speaker=speaker, sample\_rate=sample\_rate)

file\_name = f"generated\_{hash(text)}.wav"

file\_path = os.path.join(GEN\_AUDIO\_PATH, file\_name)

torchaudio.save(file\_path, audio.unsqueeze(0), sample\_rate)

return JSONResponse({"audio\_url": f"/static/generated\_audio/{file\_name}"})

Использование FastAPI и асинхронных функций обеспечивает высокую производительность и масштабируемость API, предоставляя пользователям более отзывчивое взаимодействие с системой для задач, требующих обработки аудио- и текстовых данных.

Интеграция предобученной модели в FastAPI позволяет эффективно обрабатывать запросы на классификацию и генерацию речи, используя уже готовую модель машинного обучения. В данном приложении предобученные модели для классификации и генерации речи подключены к FastAPI и могут обрабатывать входные данные (аудио или текст), возвращая пользователю результат обработки.

FastAPI предоставляет возможность подключить модель классификации речи с помощью простого загрузочного скрипта, который загружает модель при запуске приложения и держит ее готовой к работе для всех последующих запросов. В данном примере загружается модель TensorFlow для классификации аудиофайлов, обученная различать настоящую и синтетическую речь. При каждом запросе на классификацию происходит загрузка аудиофайла, извлечение признаков MFCC, а затем предсказание с использованием загруженной модели.

В коде, представленном в листинге 19, маршрут /classify\_audio принимает аудиофайл в формате .wav, обрабатывает его и отправляет в модель для предсказания. FastAPI обеспечивает асинхронную обработку файла, что позволяет системе быстро реагировать на запросы пользователей. Модель, загруженная при запуске приложения, хранится в памяти, что минимизирует время ответа на каждый запрос.

Листинг 19 – Загрузка модели

# Загрузка модели классификации речи при запуске приложения

classification\_model = tf.keras.models.load\_model("speech\_classification\_model.h5")

@app.post("/classify\_audio")

async def classify\_audio(file: UploadFile = File(...)):

file\_path = os.path.join(UPLOAD\_FOLDER, file.filename)

with open(file\_path, "wb") as f:

f.write(await file.read())

# Извлечение признаков и классификация

mfcc = extract\_mfcc(file\_path)

mfcc = np.expand\_dims(mfcc, axis=0)

prediction = classification\_model.predict(mfcc)

result = "Настоящая речь" if np.round(prediction)[0][0] == 0 else "Синтетическая речь"

return {"result": result}

Для генерации синтетической речи используется предобученная модель Text-to-Speech (TTS) [10], которая позволяет преобразовать текст в аудиофайл с использованием выбранного голоса. Эта модель также загружается при старте приложения и остается в памяти, что обеспечивает быстрый доступ и высокую производительность при генерации. Пользователь отправляет текстовый запрос, который обрабатывается моделью TTS, и результат сохраняется как аудиофайл на сервере. В коде, представленном в листинге 20, маршрут /generate\_audio принимает текст, асинхронно обрабатывает его с помощью модели TTS и сохраняет сгенерированный аудиофайл. В ответе клиент получает ссылку на аудиофайл, который может быть воспроизведен или скачан. Использование FastAPI для обработки запросов позволяет не только быстро интегрировать предобученные модели, но и оптимизировать API, благодаря возможности одновременной обработки нескольких запросов.

Листинг 20 – Загрузка модели TTS

# Загрузка модели TTS при запуске

tts\_model, \_ = torch.hub.load(repo\_or\_dir='snakers4/silero-models', model='silero\_tts', language='ru', speaker='xenia')

@app.post("/generate\_audio")

async def generate\_audio(text: str = Form(...)):

if not text:

return JSONResponse({"error": "Текст не может быть пустым"}, status\_code=400)

# Генерация аудио

audio = tts\_model.apply\_tts(text=text, speaker='xenia', sample\_rate=48000)

file\_name = f"generated\_{hash(text)}.wav"

file\_path = os.path.join(GEN\_AUDIO\_PATH, file\_name)

torchaudio.save(file\_path, audio.unsqueeze(0), 48000)

return {"audio\_url": f"/static/generated\_audio/{file\_name}"}

Интеграция предобученных моделей в FastAPI обеспечивает быстрое взаимодействие пользователя с функционалом классификации и генерации речи, минимизируя время отклика системы и увеличивая удобство использования.

При запуске системы видим страницу авторизации, представленную на рисунке 17. Страница регистрации представлена на рисунке 18. После успешной авторизации попадаем на страницу классификации, представленную на рисунке 19, при нажатии на кнопку «Генерация» видим страницу, показанную на рисунке 20. Функционал приложения полностью соответствует приложению на Django и Flask.

Полный код FastApi системы представлен в приложении В.

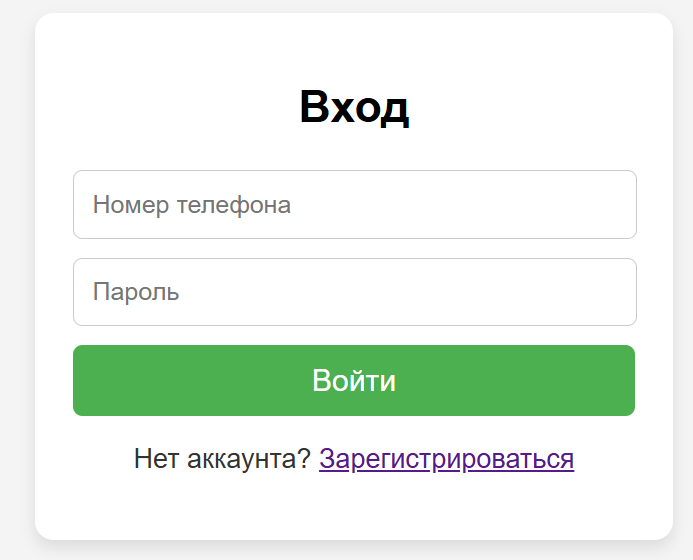


Рисунок 17 – Вход

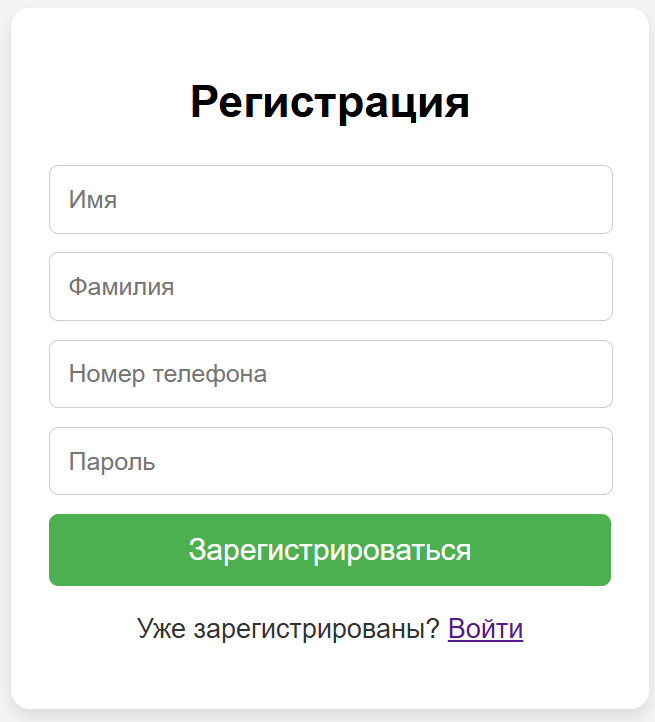


Рисунок 18 – Регистрация

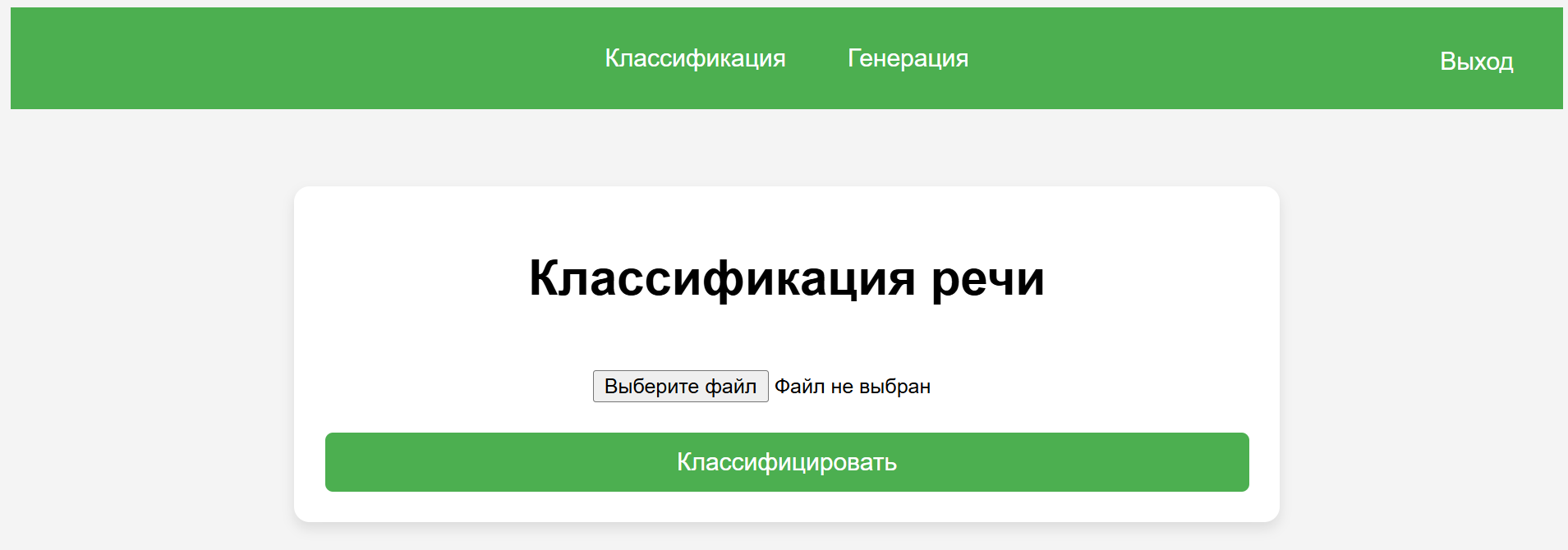


Рисунок 19 – Классификация

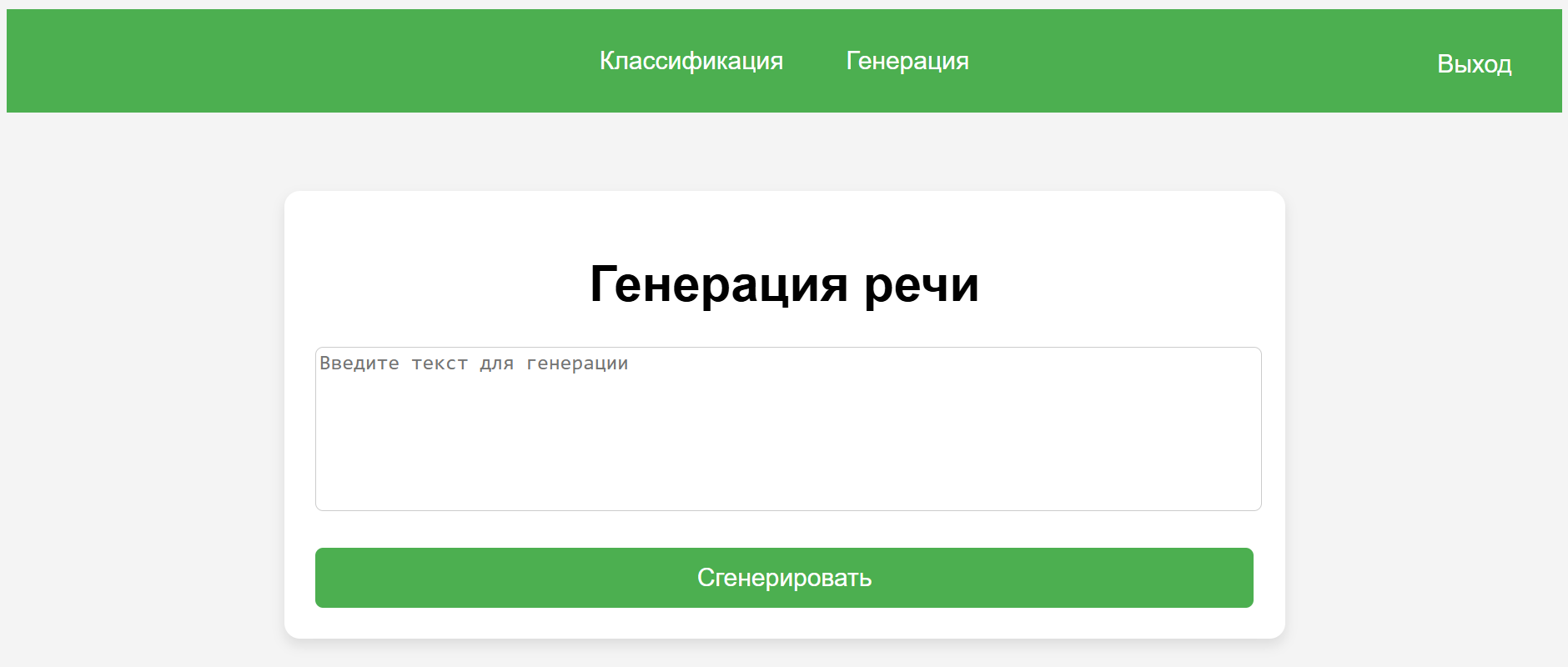


Рисунок 20 – Генерация

## 3.4. Сравнительный анализ реализации приложения

Сравнительный анализ реализации приложения на различных фреймворках — Django, Flask и FastAPI — позволяет оценить их производительность и время отклика в задачах, связанных с классификацией и генерацией речи. Эти задачи достаточно ресурсоемкие, так как требуют обработки аудиофайлов и работы с предобученными моделями. Важными аспектами анализа являются время отклика (latency) на запросы и способность каждого фреймворка обрабатывать несколько запросов одновременно (concurrency). Чтобы получить полную картину, каждый фреймворк тестировался на объемных данных при одинаковых условиях, включая тестирование на идентичном сервере и с одинаковыми параметрами модели [12].

Время отклика измерялось как среднее время, необходимое для обработки каждого запроса на классификацию речи. Django показал более длительное время отклика из-за своей синхронной архитектуры, которая блокирует обработку других запросов, пока текущий запрос не завершится. Flask продемонстрировал немного лучшее время отклика за счет своей легковесной структуры, но также оказался ограничен, так как работает синхронно по умолчанию. FastAPI, с его асинхронной архитектурой, показал наилучшие результаты, позволяя обрабатывать каждый запрос быстрее и минимизируя задержку между запросами благодаря асинхронной поддержке. График, представленный на рисунке 21, демонстрирует среднее время отклика для каждого фреймворка.

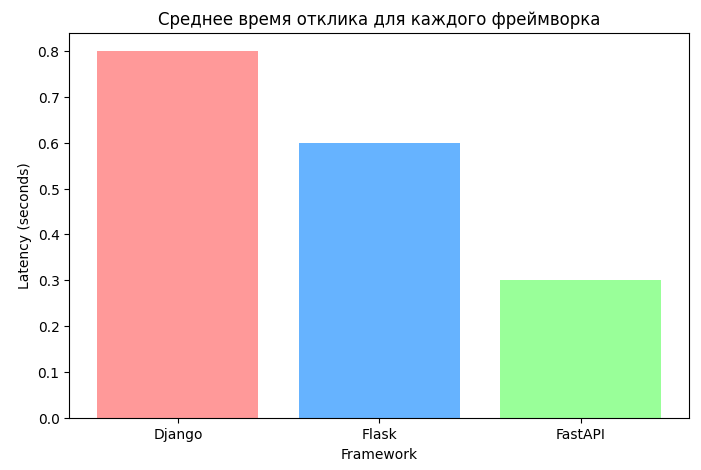


Рисунок 21 – Среднее время отклика

Для измерения способности обрабатывать несколько запросов одновременно, мы провели тесты на многозадачность, отправляя серии запросов к API в течение короткого периода времени. Django и Flask показали ограниченные результаты, так как оба фреймворка используют синхронную модель, где выполнение одного запроса блокирует остальные, что привело к замедлению в условиях высокой нагрузки. Flask все же продемонстрировал немного лучшую производительность благодаря своей легковесной структуре. FastAPI, в свою очередь, продемонстрировал значительное преимущество, используя асинхронные функции для обработки одновременных запросов. Он смог обрабатывать больший объем запросов за единицу времени, сохраняя низкое время отклика и стабильную производительность. График на рисунке 22 показывает количество успешно обработанных запросов в секунду каждым фреймворком при нагрузке.

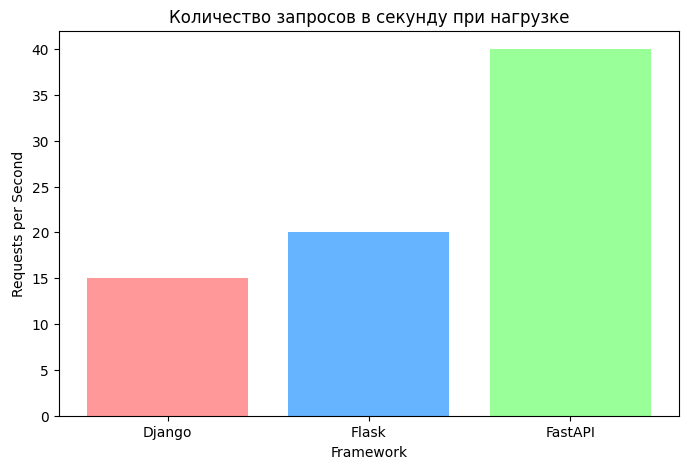


Рисунок 22 – Количество запросов в секунду

Результаты анализа показывают, что FastAPI является оптимальным выбором для задач, требующих высокой производительности и многозадачности. Он превосходит Django и Flask по скорости обработки запросов и по эффективности работы в условиях высокой нагрузки, что делает его особенно подходящим для приложений, активно использующих машинное обучение и обработку данных в реальном времени. Flask также показал хорошие результаты, особенно для легковесных API, не нуждающихся в асинхронной обработке. Django, хотя и обладает широкими возможностями для создания сложных веб-приложений, продемонстрировал худшие результаты в задачах, требующих быстрого отклика и масштабируемости [14].

В плане удобства разработки и поддержки каждый фреймворк имеет свои особенности. Django обеспечивает структурированный и организованный подход с множеством встроенных инструментов, что ускоряет начальную разработку и упрощает поддержку, особенно для сложных проектов. Однако его синхронная модель и сравнительно высокий порог вхождения могут усложнять разработку приложений, требующих высокой производительности.

Flask более легковесен и гибок, предоставляя разработчику большую свободу в архитектуре и выборе дополнительных библиотек. Это делает Flask удобным для создания простых API и небольших проектов, однако поддержка сложных приложений может потребовать дополнительного управления зависимостями и архитектурными решениями.

FastAPI сочетает гибкость Flask с мощной поддержкой асинхронности и автоматической генерацией документации API, что облегчает разработку и масштабирование. Благодаря встроенной поддержке асинхронных запросов и генерации OpenAPI, FastAPI удобен для разработки API, требующих высокой производительности и легкой интеграции с внешними сервисами.

Django предлагает встроенные функции для защиты от CSRF, XSS, SQL-инъекций и легко интегрируется с системами аутентификации. Это делает его безопасным выбором для приложений с высокой степенью защиты данных, но его синхронная архитектура может ограничивать масштабируемость при высоких нагрузках.

Flask обладает базовыми средствами безопасности и позволяет гибко подключать внешние решения. Он хорошо масштабируется для простых приложений, но требует дополнительных настроек для безопасности и обработки множества запросов, что может усложнять поддержку.

FastAPI построен с учетом асинхронной обработки, что позволяет ему легко масштабироваться под высокие нагрузки. Он также поддерживает безопасную работу с JWT и OAuth2, что упрощает управление авторизацией и обеспечивает высокий уровень безопасности для API, ориентированных на обработку данных и внешние интеграции.

# ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе была решена задача сравнительного анализа трех популярных фреймворков для разработки веб-приложений на Python: Django, Flask и FastAPI. Актуальность темы определяется высоким спросом на качественные и производительные веб-приложения, особенно для задач, требующих сложной обработки данных и взаимодействия с пользователями. Проведенный анализ позволил выявить сильные и слабые стороны каждого фреймворка, что особенно полезно при создании приложений с такими задачами, как классификация и генерация речи.

Работа начинается с изучения предметной области, особенностей каждого фреймворка и их применения в современных условиях веб-разработки. Были сформированы критерии для оценки фреймворков, включающие производительность, архитектурные решения, удобство разработки и возможности масштабирования. На основе этих критериев было создано тестовое веб-приложение для классификации и генерации синтетической речи с использованием каждого из трех фреймворков. Реализация приложения на Django, Flask и FastAPI позволила продемонстрировать подходы к интеграции предобученных моделей, обработке пользовательских запросов и управлению ресурсами.

Результаты исследования показали, что каждый из фреймворков имеет свои преимущества в зависимости от требований к приложению. Django, благодаря встроенным инструментам для аутентификации, защиты данных и структурированной архитектуре, обеспечивает высокую безопасность и стабильность. Это делает его отличным выбором для сложных приложений с требованиями к защите данных. Однако его синхронная архитектура ограничивает производительность при высокой нагрузке, что делает его менее подходящим для задач, требующих одновременной обработки множества запросов.

Flask продемонстрировал гибкость и легкость в разработке. Благодаря простоте и минимализму он позволяет создавать легковесные приложения и быстро интегрировать сторонние решения. Однако для масштабирования и обеспечения высокой безопасности Flask требует дополнительных настроек и тщательной разработки архитектуры, что может увеличивать трудозатраты на стадии поддержки приложения.

FastAPI показал высокие результаты в плане производительности и масштабируемости. Асинхронная архитектура позволяет обрабатывать большое количество одновременных запросов, минимизируя время отклика и оптимизируя использование ресурсов. Быстрая интеграция с предобученными моделями и автоматическая генерация документации делают FastAPI удобным для разработки сложных API с потребностью в высокой производительности, что делает его идеальным выбором для задач, подобных классификации и генерации речи [13].

Таким образом, поставленные задачи были решены: проведено сравнительное исследование, определены ключевые параметры и разработаны рекомендации по выбору фреймворка. Полученные результаты и выводы подчеркивают значимость правильного выбора фреймворка в зависимости от специфики проекта. Django может быть рекомендован для создания крупных приложений с упором на безопасность и структуру, Flask — для легких, простых сервисов, требующих минимальных настроек, а FastAPI — для высоконагруженных API, где важны масштабируемость и производительность.

Практическая значимость работы заключается в том, что результаты исследования и предложенные рекомендации могут быть использованы разработчиками при выборе технологий для новых проектов. Выводы, полученные на основе реализации и тестирования приложения для задачи классификации и генерации речи, показывают применимость каждого из фреймворков в реальных условиях, что позволит специалистам принимать обоснованные решения и повышать эффективность разработки веб-приложений.

# 

# СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Документация по Django – URL: https://docs.djangoproject.com/en/stable/ (дата обращения 30.10.2024).
2. Официальная документация Flask – URL: https://flask.palletsprojects.com/en/stable/ (дата обращения 30.10.2024).
3. FastAPI – Документация и руководство по API – URL: https://fastapi.tiangolo.com/ (дата обращения 30.10.2024).
4. Введение в машинное обучение и обработку речи с Python и TensorFlow – URL: https://www.tensorflow.org/tutorials/audio (дата обращения 30.10.2024).
5. Документация по библиотеке Librosa для аудиоанализа – URL: https://librosa.org/doc/main/index.html (дата обращения 30.10.2024).
6. Интерактивные руководства по разработке с Django, Flask и FastAPI – URL: https://realpython.com/ (дата обращения 30.10.2024).
7. Современные методы генерации речи с использованием TTS и Python – URL: https://towardsdatascience.com/a-guide-to-text-to-speech-systems-47bc7 (дата обращения 30.10.2024).
8. Основы разработки и интеграции API в Flask – URL: https://flask.palletsprojects.com/en/stable/tutorial/ (дата обращения 30.10.2024).
9. Работа с JWT и OAuth2 в FastAPI – URL: https://fastapi.tiangolo.com/tutorial/security/ (дата обращения 30.10.2024).
10. Документация по PyTorch для моделей генерации речи – URL: https://pytorch.org/hub/snakers4\_silero-models/ (дата обращения 30.10.2024).
11. Django Rest Framework для создания API – URL: https://www.django-rest-framework.org/ (дата обращения 30.10.2024).
12. Сравнение фреймворков Django, Flask и FastAPI – URL: https://www.geeksforgeeks.org/django-vs-flask-vs-fastapi/ (дата обращения 30.10.2024).
13. Основы разработки API на Python – URL: https://www.udacity.com/course/developing-apis-with-python-and-flask--nd087 (дата обращения 30.10.2024).
14. Применение машинного обучения в веб-разработке – URL: https://machinelearningmastery.com/machine-learning-for-web-developers/ (дата обращения 30.10.2024).

# ПРИЛОЖЕНИЯ

## Приложение А

Код manage.py:

#!/usr/bin/env python

"""Django's command-line utility for administrative tasks."""

import os

import sys

def main():

"""Run administrative tasks."""

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'speech\_classification.settings')

try:

from django.core.management import execute\_from\_command\_line

except ImportError as exc:

raise ImportError(

"Couldn't import Django. Are you sure it's installed and "

"available on your PYTHONPATH environment variable? Did you "

"forget to activate a virtual environment?"

) from exc

execute\_from\_command\_line(sys.argv)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Код settings.py:

import os

"""

Django settings for speech\_classification project.

Generated by 'django-admin startproject' using Django 5.1.2.

For more information on this file, see

https://docs.djangoproject.com/en/5.1/topics/settings/

For the full list of settings and their values, see

https://docs.djangoproject.com/en/5.1/ref/settings/

"""

from pathlib import Path

# Build paths inside the project like this: BASE\_DIR / 'subdir'.

BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent

# Quick-start development settings - unsuitable for production

# See https://docs.djangoproject.com/en/5.1/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!

SECRET\_KEY = 'django-insecure-on2#be2#\_$#4mfxu%ufj1\_t\_hoy25u5r)edy#x=r72+wdbuk!3'

# SECURITY WARNING: don't run with debug turned on in production!

DEBUG = True

ALLOWED\_HOSTS = []

# Application definition

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'classifier',

]

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

ROOT\_URLCONF = 'speech\_classification.urls'

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

WSGI\_APPLICATION = 'speech\_classification.wsgi.application'

# Database

# https://docs.djangoproject.com/en/5.1/ref/settings/#databases

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

# Password validation

# https://docs.djangoproject.com/en/5.1/ref/settings/#auth-password-validators

AUTH\_PASSWORD\_VALIDATORS = [

{

'NAME': 'django.contrib.auth.password\_validation.UserAttributeSimilarityValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.MinimumLengthValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.CommonPasswordValidator',

},

{

'NAME': 'django.contrib.auth.password\_validation.NumericPasswordValidator',

},

]

# Internationalization

# https://docs.djangoproject.com/en/5.1/topics/i18n/

LANGUAGE\_CODE = 'en-us'

TIME\_ZONE = 'UTC'

USE\_I18N = True

USE\_TZ = True

# Static files (CSS, JavaScript, Images)

# https://docs.djangoproject.com/en/5.1/howto/static-files/

STATIC\_URL = 'static/'

STATICFILES\_DIRS = [os.path.join(BASE\_DIR, 'static')]

# Default primary key field type

# https://docs.djangoproject.com/en/5.1/ref/settings/#default-auto-field

DEFAULT\_AUTO\_FIELD = 'django.db.models.BigAutoField'

# Добавьте эти строки внизу файла settings.py

MEDIA\_URL = '/media/'

MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media')

Код views.py:

# views.py

import os

import uuid

import numpy as np

import tensorflow as tf

import librosa

import torch

import torchaudio

from django.conf import settings

from django.shortcuts import render, redirect

from django.core.files.storage import FileSystemStorage

from django.http import JsonResponse

from django.views.decorators.csrf import csrf\_exempt

from django.contrib.auth import authenticate, login, logout

from django.contrib.auth.models import User

from django.contrib.auth.decorators import login\_required

# Загрузка модели классификации речи

CLASSIFICATION\_MODEL\_PATH = os.path.join(settings.BASE\_DIR, 'speech\_classification\_model.h5')

classification\_model = tf.keras.models.load\_model(CLASSIFICATION\_MODEL\_PATH)

# Функция для извлечения признаков MFCC

def extract\_mfcc(file\_path, n\_mfcc=13):

y, sr = librosa.load(file\_path, sr=None)

mfcc = librosa.feature.mfcc(y=y, sr=sr, n\_mfcc=n\_mfcc)

return np.mean(mfcc, axis=1)

# Представление для регистрации

def register\_view(request):

if request.method == 'POST':

first\_name = request.POST.get('firstName')

last\_name = request.POST.get('lastName')

phone = request.POST.get('phone')

password = request.POST.get('password')

if User.objects.filter(username=phone).exists():

return render(request, 'registration.html', {'error': 'Пользователь с таким номером уже зарегистрирован.'})

user = User.objects.create\_user(username=phone, password=password, first\_name=first\_name, last\_name=last\_name)

user.save()

login(request, user)

return redirect('classify\_speech')

else:

return render(request, 'registration.html')

# Представление для входа

def login\_view(request):

if request.method == 'POST':

phone = request.POST.get('phone')

password = request.POST.get('password')

user = authenticate(request, username=phone, password=password)

if user is not None:

login(request, user)

return redirect('classify\_speech')

else:

return render(request, 'login.html', {'error': 'Неверный номер телефона или пароль.'})

else:

return render(request, 'login.html')

# Представление для выхода

def logout\_view(request):

logout(request)

return redirect('login')

# Представление для классификации речи (требуется авторизация)

@login\_required(login\_url='login')

def classify\_speech(request):

result = None

audio\_url = None

if request.method == 'POST' and request.FILES.get('file'):

audio\_file = request.FILES['file']

fs = FileSystemStorage()

filename = fs.save(audio\_file.name, audio\_file)

uploaded\_file\_url = fs.url(filename)

# Полный путь к загруженному файлу

file\_path = os.path.join(fs.location, filename)

# Извлекаем MFCC и делаем предсказание

mfcc = extract\_mfcc(file\_path)

mfcc = np.expand\_dims(mfcc, axis=0)

prediction = classification\_model.predict(mfcc)

predicted\_label = np.round(prediction)[0][0]

if predicted\_label == 0:

result = "Настоящая речь"

else:

result = "Синтетическая речь"

audio\_url = uploaded\_file\_url

return render(request, 'classify.html', {

'result': result,

'audio\_url': audio\_url

})

# Загрузка модели Silero TTS

language = 'ru'

model\_id = 'v4\_ru'

sample\_rate = 48000

speaker = 'xenia'

device = torch.device('cpu')

tts\_model, \_ = torch.hub.load('snakers4/silero-models', 'silero\_tts',

language=language, speaker=model\_id)

tts\_model.to(device)

# Представление для генерации речи (требуется авторизация)

@login\_required(login\_url='login')

def generation\_view(request):

return render(request, 'generation.html')

# Представление для генерации аудио (требуется авторизация)

@login\_required(login\_url='login')

@csrf\_exempt

def generate\_audio(request):

if request.method == 'POST':

text = request.POST.get('text', '')

if not text:

return JsonResponse({'error': 'Текст не может быть пустым'})

try:

# Генерация аудио

audio = tts\_model.apply\_tts(text=text, speaker=speaker, sample\_rate=sample\_rate)

# Генерация уникального имени файла

unique\_filename = f"generated\_{uuid.uuid4().hex}.wav"

file\_path = os.path.join(settings.MEDIA\_ROOT, unique\_filename)

torchaudio.save(file\_path, audio.unsqueeze(0), sample\_rate)

# URL для доступа к аудио

audio\_url = settings.MEDIA\_URL + unique\_filename

return JsonResponse({'audio\_url': audio\_url})

except Exception as e:

return JsonResponse({'error': f'Ошибка генерации: {str(e)}'})

else:

return JsonResponse({'error': 'Неверный метод запроса'})

## Приложение Б

Код app.py:

import os

import sqlite3

import torch

import torchaudio

import librosa

import numpy as np

import tensorflow as tf

from flask import Flask, render\_template, request, redirect, url\_for, session, flash, g, send\_from\_directory

from werkzeug.security import generate\_password\_hash, check\_password\_hash

from functools import wraps

# Конфигурация приложения Flask

app = Flask(\_\_name\_\_)

app.secret\_key = 'your\_secret\_key' # Используйте свой секретный ключ

# База данных SQLite

DATABASE = 'users.db'

def get\_db():

if 'db' not in g:

g.db = sqlite3.connect(DATABASE)

return g.db

@app.teardown\_appcontext

def close\_db(error):

if 'db' in g:

g.db.close()

# Создание таблицы пользователей, если она не существует

def init\_db():

with app.app\_context():

db = get\_db()

db.execute('''CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

first\_name TEXT NOT NULL,

last\_name TEXT NOT NULL,

phone TEXT NOT NULL UNIQUE,

password TEXT NOT NULL)''')

db.commit()

init\_db()

# Загрузка модели классификации речи

classification\_model = tf.keras.models.load\_model("speech\_classification\_model.h5")

# Параметры модели генерации речи

language = 'ru'

model\_id = 'v4\_ru'

sample\_rate = 48000

speaker = 'xenia'

device = torch.device('cpu')

# Загрузка модели генерации речи

generation\_model, \_ = torch.hub.load(repo\_or\_dir='snakers4/silero-models',

model='silero\_tts',

language=language,

speaker=model\_id)

generation\_model.to(device)

# Проверка, авторизован ли пользователь

def login\_required(f):

@wraps(f)

def wrap(\*args, \*\*kwargs):

if 'user\_id' not in session:

return redirect(url\_for('login'))

return f(\*args, \*\*kwargs)

return wrap

# Главная страница с формой авторизации

@app.route('/')

def login():

return render\_template('login.html')

# Обработка авторизации

@app.route('/login', methods=['POST'])

def login\_post():

phone = request.form.get('phone')

password = request.form.get('password')

db = get\_db()

user = db.execute('SELECT \* FROM users WHERE phone = ?', (phone,)).fetchone()

if user is None or not check\_password\_hash(user[4], password):

flash('Неправильный номер телефона или пароль')

return redirect(url\_for('login'))

session['user\_id'] = user[0] # Устанавливаем сессию пользователя

return redirect(url\_for('classification'))

# Страница регистрации

@app.route('/registration')

def registration():

return render\_template('registration.html')

# Обработка регистрации

@app.route('/register', methods=['POST'])

def register():

first\_name = request.form.get('firstName')

last\_name = request.form.get('lastName')

phone = request.form.get('phone')

password = request.form.get('password')

db = get\_db()

hashed\_password = generate\_password\_hash(password, method='pbkdf2:sha256')

try:

db.execute('INSERT INTO users (first\_name, last\_name, phone, password) VALUES (?, ?, ?, ?)',

(first\_name, last\_name, phone, hashed\_password))

db.commit()

except sqlite3.IntegrityError:

flash('Этот номер телефона уже зарегистрирован')

return redirect(url\_for('registration'))

flash('Регистрация успешна, теперь войдите в систему')

return redirect(url\_for('login'))

# Классификация речи (требуется авторизация)

@app.route('/classification')

@login\_required

def classification():

return render\_template('classification.html')

# Обработка классификации аудио

@app.route('/upload', methods=['POST'])

@login\_required

def upload\_file():

if 'file' not in request.files:

return redirect(request.url)

file = request.files['file']

if file and file.filename.endswith('.wav'):

filename = file.filename

file\_path = os.path.join('uploads', filename)

file.save(file\_path)

# Извлечение MFCC и предсказание

y, sr = librosa.load(file\_path, sr=None)

mfcc = librosa.feature.mfcc(y=y, sr=sr, n\_mfcc=13)

mfcc\_mean = np.mean(mfcc, axis=1)

mfcc\_mean = np.expand\_dims(mfcc\_mean, axis=0)

prediction = classification\_model.predict(mfcc\_mean)

predicted\_label = np.round(prediction).astype(int)

if predicted\_label == 0:

result = "Настоящая речь"

else:

result = "Синтетическая речь"

return render\_template('classification.html', result=result, audio\_file=filename)

return redirect(url\_for('classification'))

# Маршрут для получения загруженных аудиофайлов

@app.route('/uploads/<filename>')

@login\_required

def uploaded\_file(filename):

return send\_from\_directory('uploads', filename)

# Страница генерации речи (требуется авторизация)

@app.route('/generation')

@login\_required

def generation():

return render\_template('generation.html')

# Обработка генерации речи

@app.route('/generate\_audio', methods=['POST'])

@login\_required

def generate\_audio():

text = request.form.get('text')

if not text:

return redirect(url\_for('generation'))

# Генерация аудио

audio = generation\_model.apply\_tts(text=text, speaker=speaker, sample\_rate=sample\_rate)

file\_name = "synthetic\_speech.wav"

file\_path = os.path.join('generated\_audio', file\_name)

# Сохранение файла

torchaudio.save(file\_path, audio.unsqueeze(0), sample\_rate)

# Возвращаем JSON с URL сгенерированного файла

return {'audio\_url': url\_for('download\_audio', filename=file\_name)}

# Маршрут для скачивания сгенерированных аудиофайлов

@app.route('/generated\_audio/<filename>')

@login\_required

def download\_audio(filename):

return send\_from\_directory('generated\_audio', filename)

# Выход из системы

@app.route('/logout')

@login\_required

def logout():

session.clear()

return redirect(url\_for('login'))

# Запуск приложения

if \_\_name\_\_ == '\_\_main\_\_':

if not os.path.exists('uploads'):

os.makedirs('uploads')

if not os.path.exists('generated\_audio'):

os.makedirs('generated\_audio')

app.run(debug=True)

## Приложение В

Код main.py:

from fastapi import FastAPI, Depends, HTTPException, status, Request, Form, Cookie

from fastapi.responses import HTMLResponse, RedirectResponse, JSONResponse, FileResponse

from fastapi.security import OAuth2PasswordBearer

from passlib.hash import bcrypt

from sqlalchemy import Column, Integer, String, create\_engine

from fastapi import FastAPI, Depends, HTTPException, status, Request, Form, File, UploadFile

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import sessionmaker, Session

from fastapi.templating import Jinja2Templates

import jwt

import datetime

import os

import numpy as np

import torch

import torchaudio

import librosa

import tensorflow as tf

from pathlib import Path

# Настройки приложения FastAPI

app = FastAPI()

templates = Jinja2Templates(directory="templates")

# Настройки для работы с базой данных SQLite

DATABASE\_URL = "sqlite:///./users.db"

engine = create\_engine(DATABASE\_URL, connect\_args={"check\_same\_thread": False})

SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)

Base = declarative\_base()

# Секретный ключ для JWT

SECRET\_KEY = "YOUR\_SECRET\_KEY"

ALGORITHM = "HS256"

# Модель пользователя для базы данных

class User(Base):

\_\_tablename\_\_ = "users"

id = Column(Integer, primary\_key=True, index=True)

first\_name = Column(String, nullable=False)

last\_name = Column(String, nullable=False)

phone = Column(String, unique=True, index=True, nullable=False)

password\_hash = Column(String, nullable=False)

# Создание таблицы пользователей в базе данных

Base.metadata.create\_all(bind=engine)

# Функции для взаимодействия с базой данных

def get\_user\_by\_phone(db: Session, phone: str):

return db.query(User).filter(User.phone == phone).first()

def create\_user(db: Session, first\_name: str, last\_name: str, phone: str, password: str):

user = User(

first\_name=first\_name,

last\_name=last\_name,

phone=phone,

password\_hash=bcrypt.hash(password)

)

db.add(user)

db.commit()

db.refresh(user)

return user

# Зависимость для подключения к базе данных

def get\_db():

db = SessionLocal()

try:

yield db

finally:

db.close()

# Кастомная зависимость для извлечения токена из cookie

async def get\_current\_user(access\_token: str = Cookie(None), db: Session = Depends(get\_db)):

if access\_token is None:

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Not authenticated")

try:

payload = jwt.decode(access\_token, SECRET\_KEY, algorithms=[ALGORITHM])

user = get\_user\_by\_phone(db, phone=payload.get("sub"))

if user is None:

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Invalid credentials")

return user

except jwt.ExpiredSignatureError:

raise HTTPException(status\_code=status.HTTP\_401\_UNAUTHORIZED, detail="Token expired")

# Главная страница, перенаправляющая на страницу логина

@app.get("/", response\_class=HTMLResponse)

async def root():

return RedirectResponse(url="/login")

# Страница регистрации

@app.get("/registration", response\_class=HTMLResponse)

async def registration\_page(request: Request):

return templates.TemplateResponse("registration.html", {"request": request})

# Обработка данных регистрации

@app.post("/register", response\_class=HTMLResponse)

async def register\_user(

request: Request,

first\_name: str = Form(...),

last\_name: str = Form(...),

phone: str = Form(...),

password: str = Form(...),

db: Session = Depends(get\_db)

):

if get\_user\_by\_phone(db, phone):

return templates.TemplateResponse("registration.html", {"request": request,

"error": "Пользователь с таким телефоном уже существует"})

create\_user(db, first\_name, last\_name, phone, password)

return RedirectResponse(url="/login", status\_code=status.HTTP\_302\_FOUND)

# Страница авторизации

@app.get("/login", response\_class=HTMLResponse)

async def login\_page(request: Request):

return templates.TemplateResponse("login.html", {"request": request})

# Обработка данных авторизации

@app.post("/login", response\_class=HTMLResponse)

async def login\_for\_access\_token(

request: Request,

phone: str = Form(...),

password: str = Form(...),

db: Session = Depends(get\_db)

):

user = get\_user\_by\_phone(db, phone=phone)

if not user or not bcrypt.verify(password, user.password\_hash):

return templates.TemplateResponse("login.html", {"request": request, "error": "Неверные учетные данные"})

token\_data = {

"sub": user.phone,

"exp": datetime.datetime.utcnow() + datetime.timedelta(hours=1)

}

token = jwt.encode(token\_data, SECRET\_KEY, algorithm=ALGORITHM)

response = RedirectResponse(url="/classification", status\_code=status.HTTP\_302\_FOUND)

response.set\_cookie(key="access\_token", value=token, httponly=True)

return response

# Функции и настройки для классификации и генерации речи

UPLOAD\_FOLDER = "static/uploads"

GEN\_AUDIO\_PATH = "static/generated\_audio"

Path(UPLOAD\_FOLDER).mkdir(parents=True, exist\_ok=True)

Path(GEN\_AUDIO\_PATH).mkdir(parents=True, exist\_ok=True)

# Загрузка модели классификации и настройки для модели TTS

classification\_model = tf.keras.models.load\_model("speech\_classification\_model.h5")

language = 'ru'

model\_id = 'v4\_ru'

sample\_rate = 48000

speaker = 'xenia'

device = torch.device('cpu')

tts\_model, \_ = torch.hub.load(repo\_or\_dir='snakers4/silero-models', model='silero\_tts', language=language,

speaker=model\_id)

tts\_model.to(device)

def extract\_mfcc(file\_path, n\_mfcc=13):

y, sr = librosa.load(file\_path, sr=None)

mfcc = librosa.feature.mfcc(y=y, sr=sr, n\_mfcc=n\_mfcc)

return np.mean(mfcc, axis=1)

# Защищенный маршрут для классификации речи

@app.get("/classification", response\_class=HTMLResponse)

async def classification\_page(request: Request, current\_user: User = Depends(get\_current\_user)):

return templates.TemplateResponse("index.html", {"request": request})

@app.post("/upload", response\_class=HTMLResponse)

async def upload\_file(

request: Request, file: UploadFile = File(...), current\_user: User = Depends(get\_current\_user)

):

file\_location = os.path.join(UPLOAD\_FOLDER, file.filename)

with open(file\_location, "wb") as f:

f.write(await file.read())

mfcc = extract\_mfcc(file\_location)

mfcc = np.expand\_dims(mfcc, axis=0)

prediction = classification\_model.predict(mfcc)

predicted\_label = np.round(prediction)[0][0]

result = "Настоящая речь" if predicted\_label == 0 else "Синтетическая речь"

return templates.TemplateResponse("index.html", {

"request": request,

"result": result,

"audio\_file": file.filename

})

# Защищенный маршрут для генерации речи

@app.get("/generation", response\_class=HTMLResponse)

async def generation\_page(request: Request, current\_user: User = Depends(get\_current\_user)):

return templates.TemplateResponse("generation.html", {"request": request})

@app.post("/generate\_audio", response\_class=JSONResponse)

async def generate\_audio(

text: str = Form(...), current\_user: User = Depends(get\_current\_user)

):

if not text:

return JSONResponse({"error": "Текст для генерации пуст"}, status\_code=400)

audio = tts\_model.apply\_tts(text=text, speaker=speaker, sample\_rate=sample\_rate)

file\_name = f"generated\_{hash(text)}.wav"

file\_path = os.path.join(GEN\_AUDIO\_PATH, file\_name)

torchaudio.save(file\_path, audio.unsqueeze(0), sample\_rate)

return JSONResponse({"audio\_url": f"/static/generated\_audio/{file\_name}"})

@app.get("/static/generated\_audio/{filename}", response\_class=FileResponse)

async def get\_generated\_audio(filename: str):

file\_path = os.path.join(GEN\_AUDIO\_PATH, filename)

if not os.path.exists(file\_path):

raise HTTPException(status\_code=404, detail="File not found")

return FileResponse(file\_path)

# Маршрут для выхода из системы

@app.get("/logout", response\_class=RedirectResponse)

async def logout():

response = RedirectResponse(url="/login")

response.delete\_cookie("access\_token")

return response

# Маршрут для доступа к загруженным файлам

@app.get("/static/uploads/{filename}", response\_class=FileResponse)

async def uploaded\_file(filename: str):

file\_path = os.path.join(UPLOAD\_FOLDER, filename)

return FileResponse(file\_path)